



# Implementing True Serializable Transactions

THE PYRRHO V7 EXPERIMENT  
MALCOLM CROWE, FRITZ LAUX

This tutorial is about DBMS implementation, and serializable transactions. Specifically, this experiment looks at whether the extensive use of shareable data structures with a full-featured relational DBMS implementation is feasible. Experiments reported to previous DBKDA conferences seemed to indicate that such a path was possible, but it was not clear how practical it would be.

# Malcolm Crowe



- ▶ Malcolm Crowe is an Emeritus Professor at the University of the West of Scotland, where he worked from 1972 (when it was Paisley College of Technology) until 2018.
- ▶ He gained a D.Phil. in Mathematics at the University of Oxford in 1979.
- ▶ He was appointed head of the Department of Computing in 1985. His funded research projects before 2001. were on Programming Languages and Cooperative Work.
- ▶ Since 2001 he has worked steadily on PyrrhoDBMS to explore optimistic technologies for relational databases and this work led to involvement in DBTech, and a series of papers and other contributions at IARIA conferences with Fritz Laux, Martti Laiho, and others.
- ▶ Prof. Crowe has recently been appointed an IARIA Fellow.

2

I'm Malcolm Crowe, a retired professor from the University of the West of Scotland. I started off in Mathematics, transferred to Computing in 1979, and I have been working on database implementation since 2001.

## Fritz Laux



- ▶ Prof. Dr. Fritz Laux was professor (now emeritus) for Database and Information Systems at Reutlingen University from 1986 - 2015. He holds an MSc (Diplom) and PhD (Dr. rer. nat.) in Mathematics.
- ▶ His research focuses on database modeling and technology, transaction processing, data warehousing and data analytics. He has published a number of papers in peer reviewed conferences and journals on the above topics, some of them have received Best Paper Awards. He is a regular contributor and speaker at DBKDA.
- ▶ Prof. Laux is a co-founder of DBTechNet, an initiative of European universities and IT-companies to set up a transnational collaboration scheme of higher level education in Databases. Together with colleagues from 5 European countries he was conducting projects supported by the European Union on state-of-the-art teaching and hands-on labs on database technology.
- ▶ Prof. Laux received the 2012 Research Award from Reutlingen University and he is an IARIA fellow.

3

Prof. Dr Fritz Laux is a retired professor from Reutlingen University, also originally a mathematician, who was appointed Professor of Database and Information Systems in 1986. His research papers focus on database modelling and technology, transaction processing, data warehousing and data analytics, and he is a regular contributor to DBKDA.

## This tutorial aims



- ▶ To consider usability of methods that help enforce strict ACID for DBMS
- ▶ Some new ideas
- ▶ Proof of concept at least
- ▶ Our viewpoint: full isolation requires truly serializable transactions
- ▶ All the textbooks say it's important
  - ▶ But then start making excuses
- ▶ Pyrrho v7 is our second experiment

4

In this tutorial we will consider some methods that help to enforce strict atomicity, consistency, isolation and durability for database management systems. There are some new ideas and the Pyrrho experiment brings proof of concept.

Our starting point is that full isolation requires truly serializable transactions.

All database textbooks begin by saying how important serializability and isolation are, but very quickly settle for something much less. This is really our second

experiment on using shareability.

## RDBMS Implementation 1



- ▶ If we agree that ACID transactions are good
  - ▶ We should not start to write anything until commit and then write the whole transaction at once
  - ▶ We should guarantee consistency by validating every commit against the current database
  - ▶ We should enforce actual isolation by not allowing any user to see transactions in progress
  - ▶ We should enforce durability by using durable media (preferably write-once append storage)
- ▶ These should be prioritised above speed and replication

5

If we agree that ACID transactions are good, then:

First, for atomicity and durability we should not write anything durable until (a) we are sure we wish to commit and (b) we are ready to write the whole transaction.

Second: before we write anything durable, we should validate our commit against the current database.

Third, for isolation, we should not allow any user to see transactions that have not yet been committed.

Fourth, for durability, we should use durable media – preferably write-once append storage.

## RDBMS Implementation 2



- ▶ Good isolation should mean the transaction log should show non-overlapping txs
  - ▶ Can reorder them if there is no conflict
- ▶ Then it might be enough to use append storage for the database file
  - ▶ And do the reordering as we go!
- ▶ But distributed transactions are a problem
  - ▶ At most 1 remote participant in a transaction
  - ▶ Or we get the 2-army problem
  - ▶ Every piece of data has a “transaction master”

6

From the last slide, it seems clear that a database should record its durable transactions in a non-overlapping manner.

If transactions in progress overlap in time, they cannot both commit if they conflict: and if they don't conflict, it does not matter which one is recorded first. The simplest order for writing is that of the transaction commit.

If we are writing some changes that we prepared earlier, the validation step must ensure that it depends on nothing that has changed in the meantime, so that our change can seem to date from the time it was

committed rather than first considered. Effectively we need to reorder overlapping transactions as we commit them.

These few rules guarantee actual serialization of transactions for a single transaction log (sometimes called a single transaction master). It obviously does not matter where the transactions are coming from.

But if a transaction is trying to commit changes to more than one transaction log, things are very difficult. If messages between autonomous actors can get lost, then inconsistencies are inevitable. One-way dependency can be implemented with a single remote participant.

# Demo 1: Transaction Log



- ▶ In this demonstration, we will see how every Transaction  $T$  consists of a set of elementary operations  $e_1, e_2, \dots, e_n$ .
- ▶ Each operation corresponds to a Physical object written to the transaction log
- ▶ Committing this transaction applies the sequence to the Database  $D$ . In reverse mathematical notation

$$(D)T = (..((D)e_1)e_2..)e_n$$



- ▶ Every Database  $D$  is the result of applying a sequence of transactions to the empty \_system Database  $D_0$



- ▶ The sequence of transactions and their operations is recorded in the log.

7

Our first demonstration is about the transaction log. The transaction log defines the contents of the database.

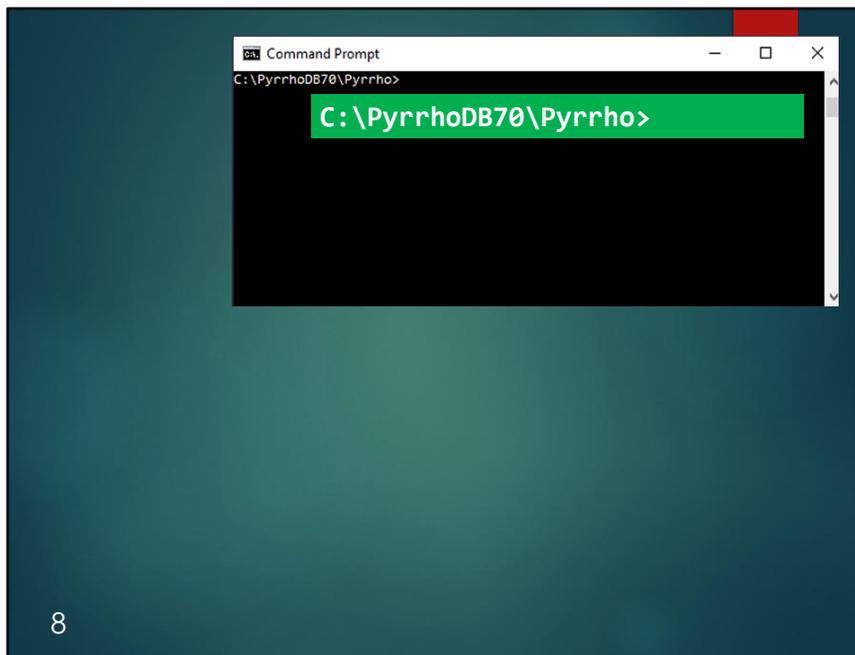
If we think of a transaction commit as comprising a set of elementary operations  $e$ , then the transaction log is best implemented as a serialization of these events to the append storage. We can think of these serialized packets as objects in the physical database. In an object-oriented programming language, we naturally have a class of such objects, and we call this class *Physical*.

So, at the commit point of a transaction, we have a list of these objects, and the commit has two effects (a)

appending them to the storage, (b) modifying the database so that other users can see.

We can think of each of these elementary operations as a transformation on the database, to be applied in the order they are written to the database (so the ordering within each transaction is preserved). And the transaction itself is the resulting transformation of the database.

Any state of the database is thus the result of the entire sequence of committed transactions, starting from a known initial database state corresponding to an empty database.

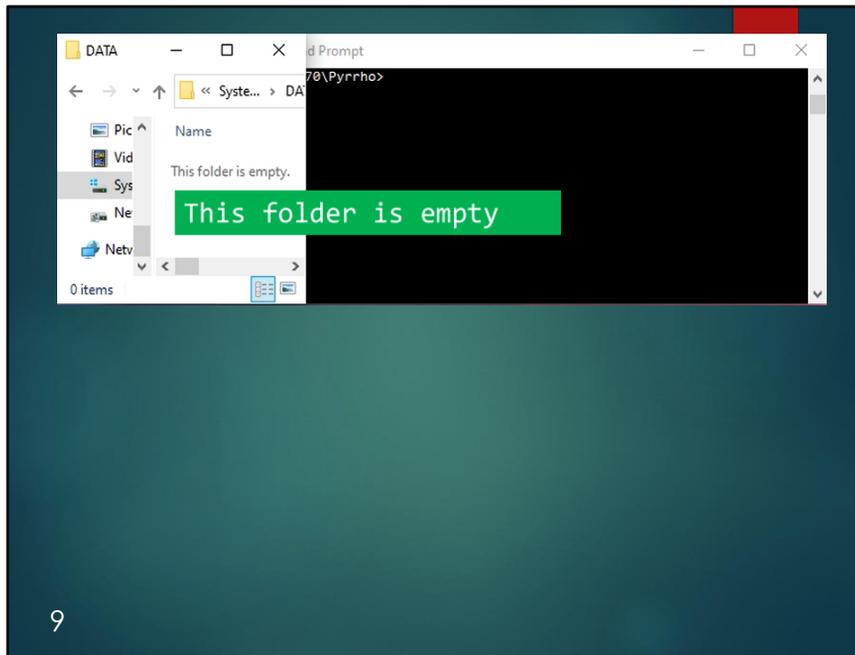


8

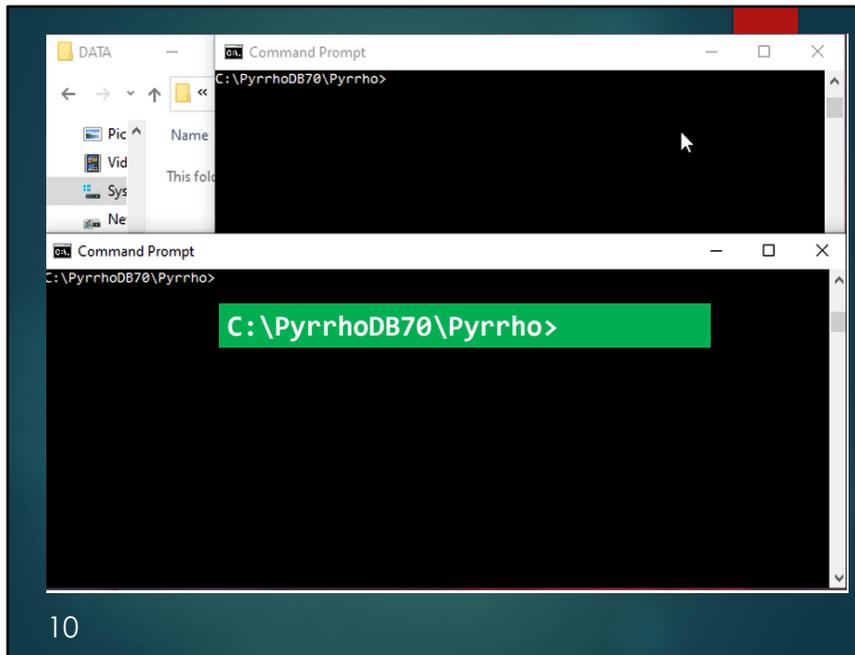
Let's start with a command window, set to whatever folder the distribution is in. Don't worry if yours is different.

The Pyrrho distribution folder contains the server and command line processor executable files, and the PyrrhoLink dll. (You can copy these files to anywhere convenient.)

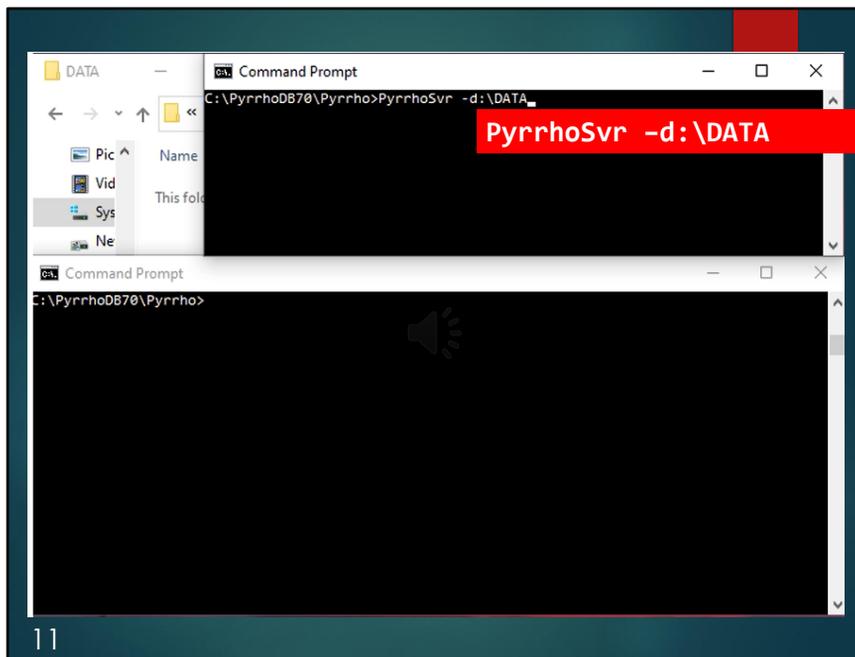
This command window will be for the server.



Let us agree on a folder for the database files. Create a new folder \DATA. To save space, as here, you can overlap the windows a bit.

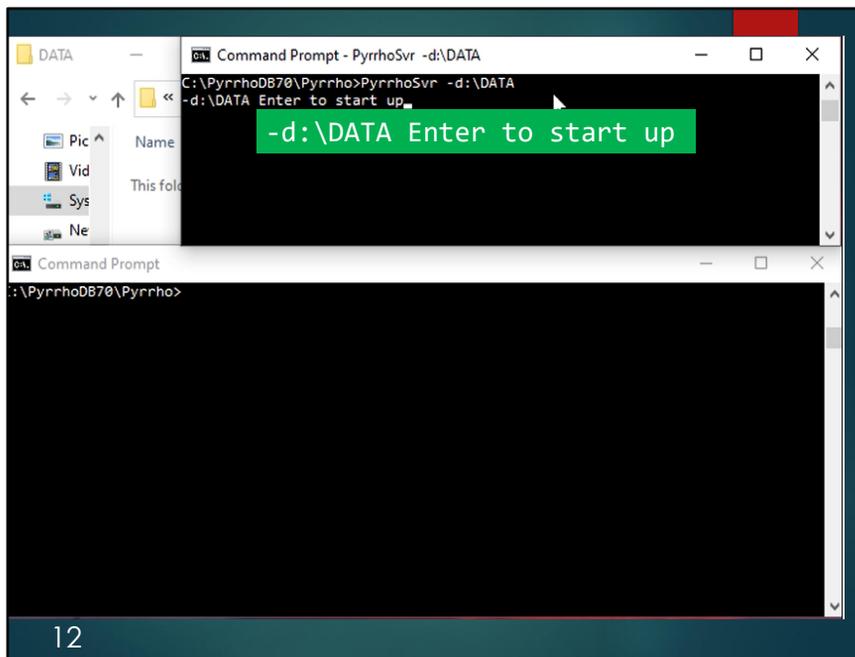


Before we start the server, let's add a command window for the client, also with the same folder. We will see it is better to make it wider than normal.

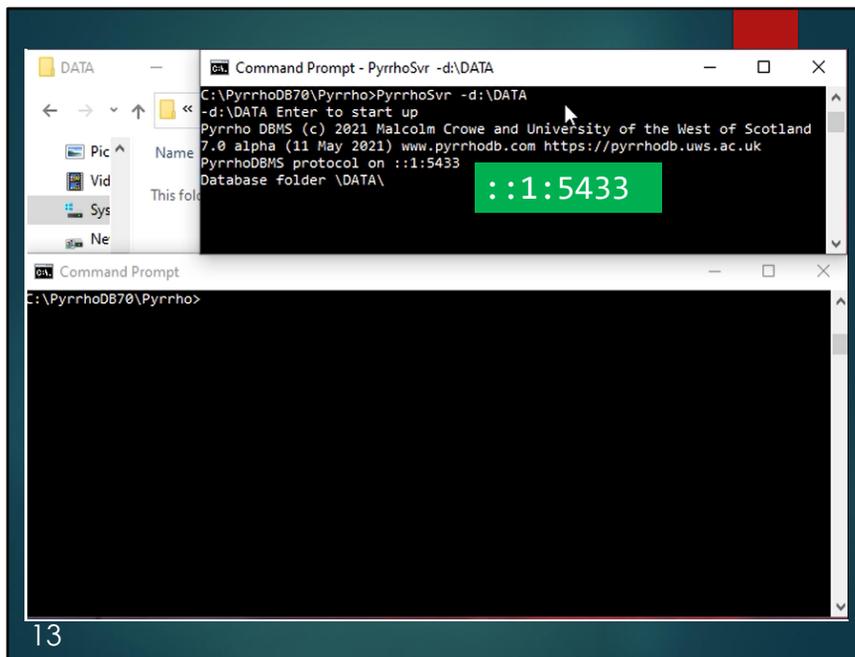


The Pyrrho server is called PyrrhoSvr, and it runs in an ordinary account with no special privileges. You can copy it to anywhere convenient.

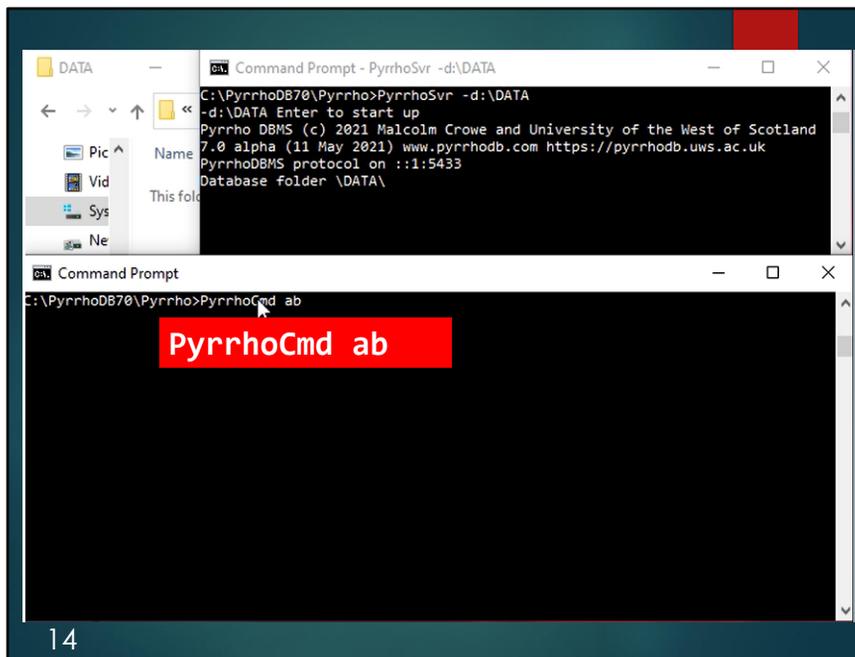
It is easiest just to use it in a command window. When we specify the server, we can provide a folder for it to store database files in.



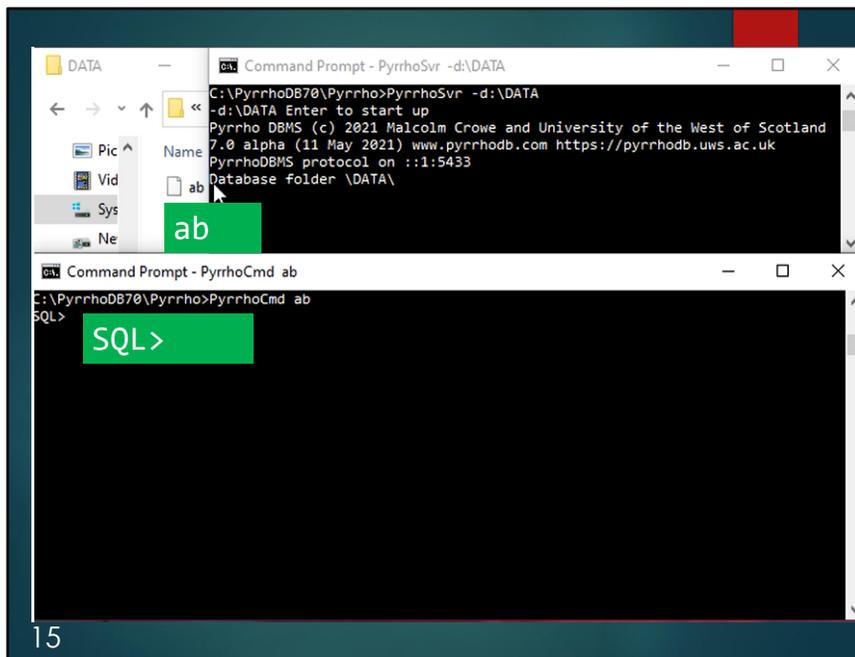
When prompted, we verify that the options specified are the ones we want, and click Enter.



You then need to leave the command window open (but of course you can minimise it). It can be useful for diagnostic information if something goes wrong. On startup, Pyrrho checks that it can access the specified folder, but places nothing in it. Databases are created by users, and the transaction log will be stored by the server on disk, usually in this default folder. It also announces its TCP/IP request port.

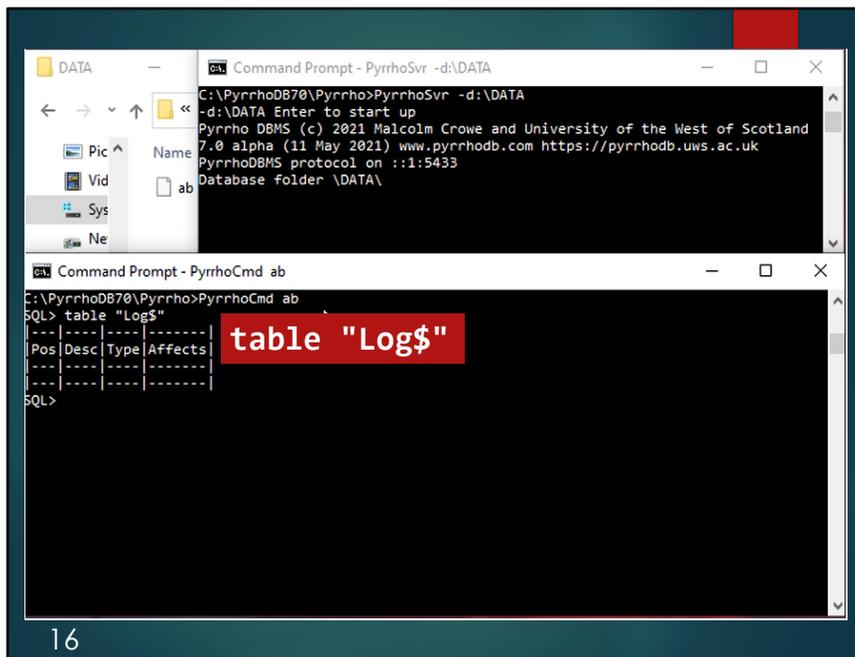


The simplest way to get Pyrrho to do work for you is to connect to it using the command line processor PyrrhoCmd, and when you do that you can specify the database you are connecting to. It can be a new database, as here.



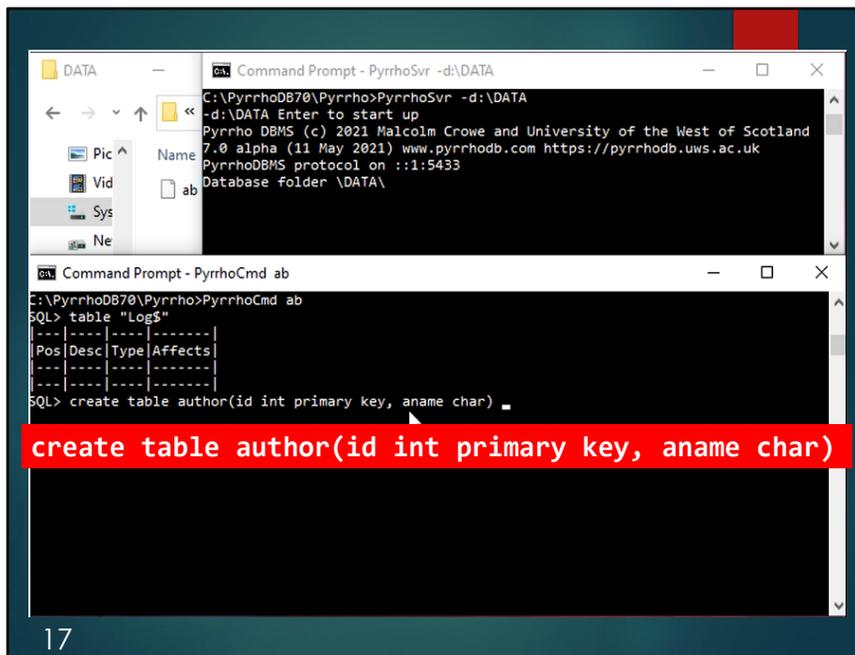
And when you give the command, the server immediately creates an empty database in the database folder.

This is a transaction log, and at the moment it contains just 5 bytes, and its opened exclusively by the server, so that we can't look at it unless the server is stopped.

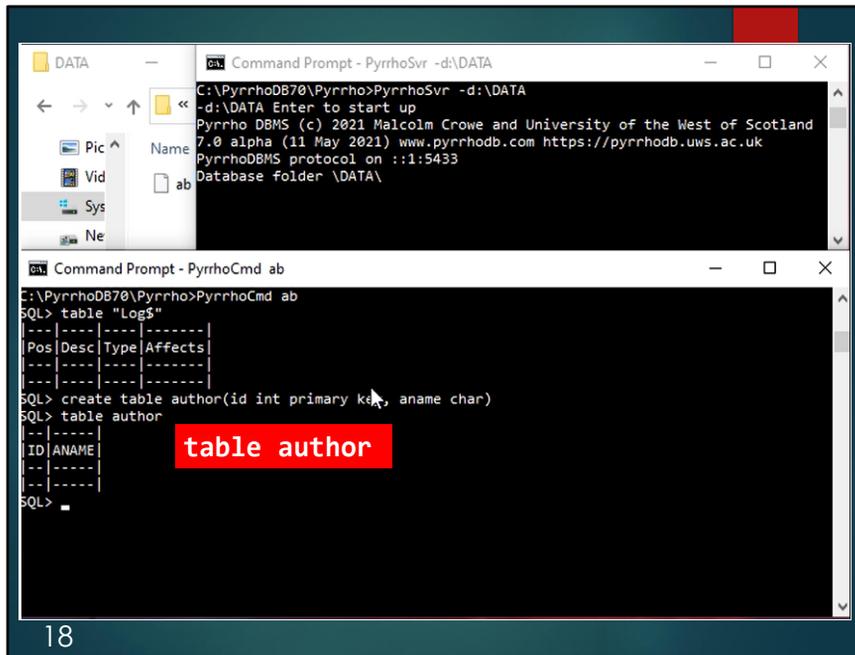


16

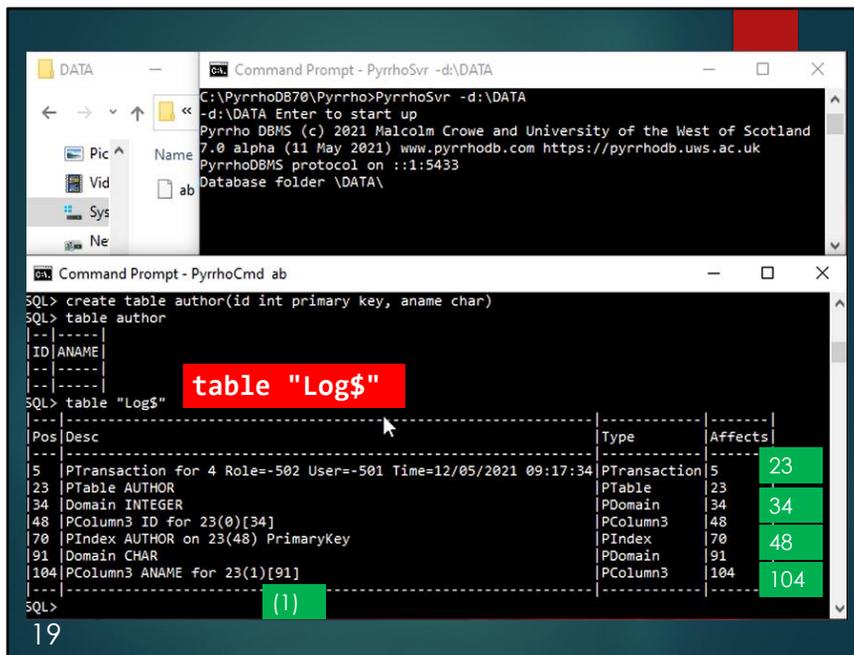
We can examine the contents of the transaction log with the table “Log\$” statement. Log\$ is one of many system tables for inspecting server internals from SQL. It is empty at the moment.



Let's make a base table in the database ab, by giving a CREATE TABLE command, and this is the normal syntax in the SQL standard, but Pyrrho has its own ideas about primitive types. So CHAR, for example, is an unbounded character string, and INT is actually a "bigint" – it's up to 2000 bits



So, we have created the table, but let's look at it: of course it will be empty.



Let's examine how this has been represented in the transaction log. We can see that there are 7 objects in the transaction log, all wrapped in a single transaction. The AUTHOR table is mentioned, and the domains of the two columns, and the two column names, and the primary key as defined in the create table request, is defined there along with its key.

You will notice that the file positions that are here (these are actual byte positions in the file) are used as unique identifiers for objects in the database. Pyrrho uses them throughout: names can be changed but the file position

of the definition can't. So here the table AUTHOR is referred to as 23, the domain INTEGER is 34, ID is 48, and that becomes the primary key, and so on. The numbers in brackets give the ordering of the columns in the table. The log shows the identity of the user who made the changes, and the role they were using, but in this database, no users have been defined yet, so these are system defaults.

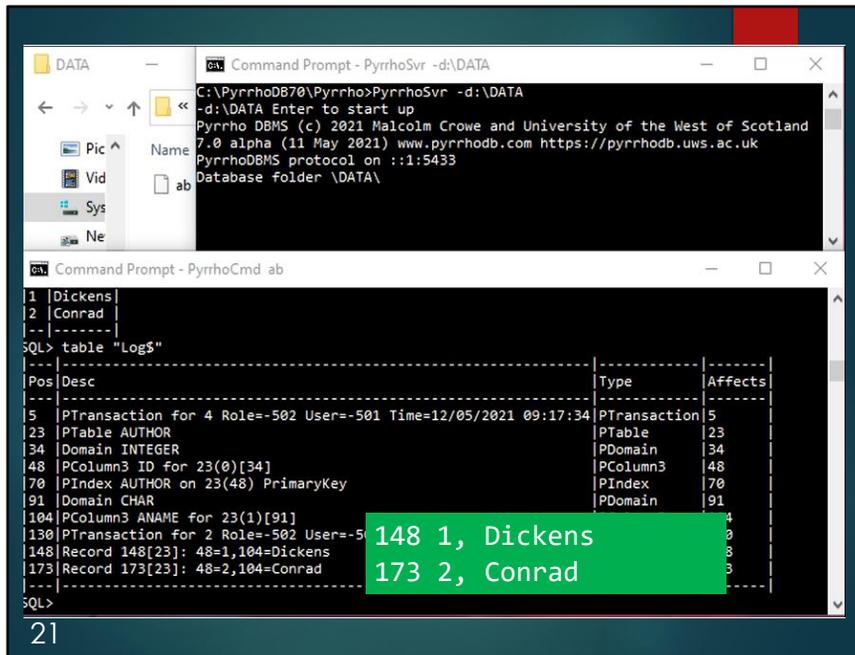
```

C:\PyrrhoDB70\Pyrrho>PyrrhoSvr -d:\DATA
-d:\DATA Enter to start up
Pyrrho DBMS (c) 2021 Malcolm Crowe and University of the West of Scotland
7.0 alpha (11 May 2021) www.pyrrhodb.com https://pyrrhodb.uws.ac.uk
PyrrhoDBMS protocol on ::1:5433
Database folder \DATA\

Command Prompt - PyrrhoCmd ab
-----
5 | PTransaction for 4 Role=-502 User=-501 Time=12/05/2021 09:17:34 | PTransaction | 5
23 | PTable AUTHOR | PTable | 23
34 | Domain INTEGER | PDomain | 34
48 | PColumn3 ID for 23(0)[34] | PColumn3 | 48
70 | PIndex AUTHOR on 23(48) PrimaryKey | PIndex | 70
91 |
104 | insert into author values(1,'Dickens'),(2,'Conrad') |
-----
SQL> insert into author values(1,'Dickens'),(2,'Conrad')
2 records affected in ab
SQL> table author
2 records affected in ab
table author
-----
ID | ANAME
-----
1 | Dickens
2 | Conrad
-----
SQL>
20

```

Let's create a couple of authors in this AUTHOR table. We can do this in a single INSERT statement. And then we display the table.



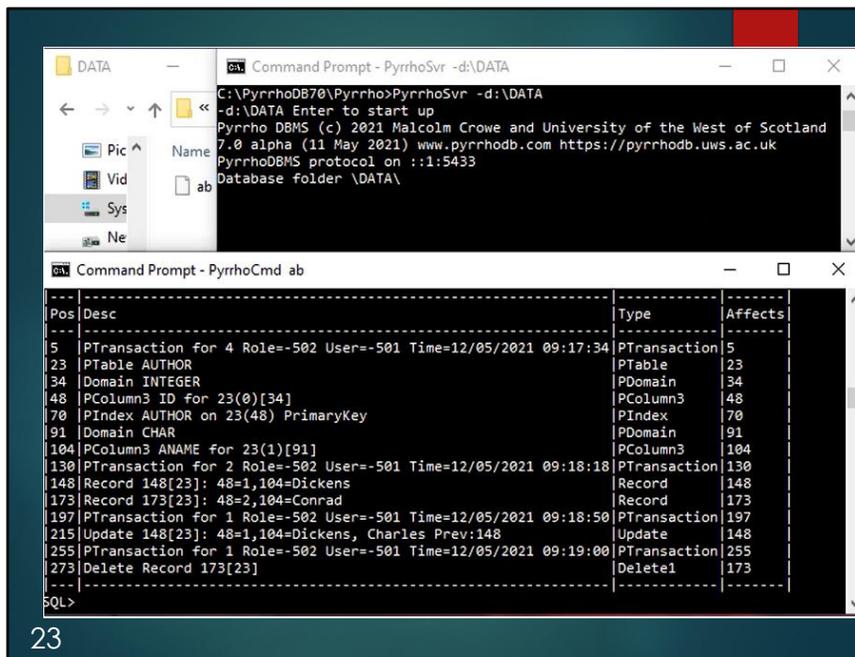
In the transaction log we see that the auto-committed transaction has both records.

```
C:\PyrrhoDB70\Pyrrho>PyrrhoSvr -d:\DATA
-d:\DATA Enter to start up
Pyrrho DBMS (c) 2021 Malcolm Crowe and University of the West of Scotland
7.0 alpha (11 May 2021) www.pyrrhodb.com https://pyrrhodb.uws.ac.uk
PyrrhoDBMS protocol on ::1:5433
Database folder \DATA\

Command Prompt - PyrrhoCmd ab
48 | PColumn3 ID for 23(0)[34] | PColumn3 | 48
70 | PIndex AUTHOR on 23(48) PrimaryKey | PIndex | 70
91 | Domain CHAR | PDomain | 91
104 | PColumn3 ANAME for 23(1)[91] | PColumn3 | 104
130 | PTransaction for 2 Role=-502 User=-501 Time=12/05/2021 09:18:18 | PTransaction | 130

update author set aname='Dickens, Charles' where id=1
SQL> update author set aname='Dickens, Charles' where id=1
1 records affected in ab
SQL> delete from author where aname='Conrad'
1 records affected in ab
SQL> table author
--|-----|
ID | ANAME
--|-----|
1 | Dickens, Charles
--|-----|
SQL>
```

We can update, and delete.



23

Finally let's see the update and delete in the Log.

This completes the demonstration showing the use of the transaction log and the transaction markers.

## RDBMS Implementation 3



- ▶ If we agree on a globalization strategy
  - ▶ Then the DBMS format should be neutral
  - ▶ Not locale or machine specific
- ▶ If we agree that arbitrary size limits are bad
  - ▶ Precision, string length etc
  - ▶ Then we need to make them huge (e.g.  $2^{64}$ )
- ▶ If we agree that security is important
  - ▶ Then we use operating system authentication
  - ▶ Don't let the user just type their credentials

24

If we agree on a globalization strategy, then the DBMS should be neutral, and not specific to a particular machine, platform, or locale. A database created on one machine, platform or locale should be usable on another.

The DBMS should not impose arbitrary size limits on strings, number of columns etc. Any that are imposed should be huge.

If we agree that security is important, then we should use operating system authentication and no other options. Users should not be simply allowed to say who

they are.

## Pyrrho DBMS



- ▶ Pyrrho DBMS always had the above goals and full feature set including triggers, views
  - ▶ But ACID's C and I were not good enough
  - ▶ For serializable transactions all RDBMS struggle
- ▶ The StrongDBMS experiment showed that using shareable data structures could help
- ▶ So we aim in Pyrrho v7 to use shareable (immutable) data structures
  - ▶ Wherever possible!

25

Pyrrho DBMS has always had these goals, and a full feature set including stored procedures, structured types, triggers, and views.

But it turned out that its consistency and isolation in its implementation were not good enough, and it was easily outperformed by StrongDBMS, a much simpler system. In an artificial test with high concurrency and serializable transactions, we found that all DBMS were outperformed by StrongDBMS.

So, in Pyrrho v7, the aim was to re-implement Pyrrho using a lesson learned from StrongDBMS, that in

situations of high transaction concurrency it was best to use shareable, immutable data structures, for as many internal structures as possible.

The implementation has been progressing steadily, it is still in at the alpha stage, but anyone can see the progress that has been made, as the source code is on github.

## Shareable data



- ▶ Strings in Java, Python, C# etc are good
  - ▶ They are immutable
  - ▶ You have to make a new string to change them
- ▶ Illegal to write `str[x]='Y'`;
- ▶ For a class to be shareable, all fields must be read-only and shareable
  - ▶ Deep initialisation needed in constructors
  - ▶ Need another constructor to make a change
- ▶ Inherited fields need to be set in the `:base()` constructor
  - ▶ Maybe with the help of a static method

26

Let's quickly review what is meant by shareable data.

Many programming languages currently have shareable implementations of strings. Specifically, strings in Java, Python and C# are immutable: if you make a change, you get a new string, and anyone who had a copy of the previous version sees no change.

In these systems, it is illegal to modify a string by assigning to a character position instead you need to use a library function. The addition operator can be used in these languages to create a sum of strings. This is basically the model for shareable data structures.

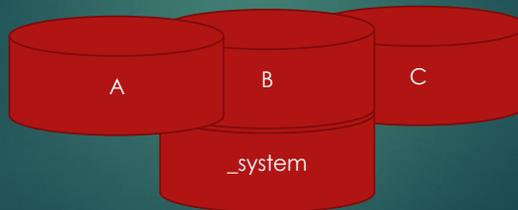
For a class to be shareable, all fields must be read-only and shareable. Constructors therefore need to perform deep initialisation, and any change to an existing structure needs another constructor.

Inherited fields need to be initialised in the base (or super) constructor.

## Share common elements



- ▶ For example, all databases share things
  - ▶ Predefined types, properties, system tables
- ▶ These don't have to be copied to a new db
  - ▶ All databases can have the same starting point



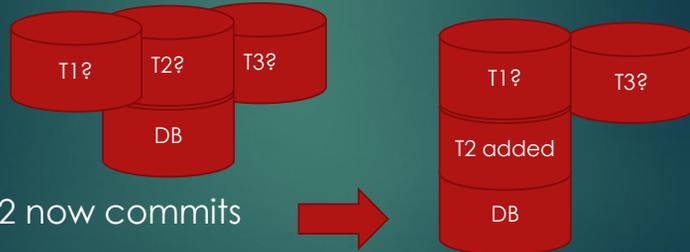
27

This is useful for databases because databases share so many things: predefined types, properties, system tables. So as mentioned on a previous slide, all databases have the same starting state.

# Transactions



- ▶ Database + possible additions, changes
  - ▶ Not shareable: contains a list of Physical
- ▶ Concurrency: several TX in progress



- ▶ T2 now commits
- ▶ T1 can commit if no conflict with T2 changes

28

Even more importantly all transactions can start with the current state of the database, without cloning or copying any internal structures. When a transaction starts, it starts with the shared database state: as it adds physicals, it transforms. Different transactions will in general start from different states of the shared database. In the picture on the slide, we know what the database state is. We don't know if any of the transactions fit on top anymore. In particular, after T2 commits, T1 and/or T3 may no longer be able to commit. If T1 was able to commit before, then it will still be able

to commit provided it has no conflict with T2's changes.

## Transaction Conflict



- ▶ The rules for this are debatable
- ▶ T1 and T2 will not conflict if they affect different tables or other database objects
  - ▶ And only read from different tables
- ▶ But we can allow them to change different rows in the same table
  - ▶ Provided they read different specified rows
- ▶ Or even different columns in the same row
  - ▶ Provided they read different columns

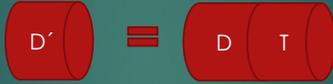
29

The details of what constitutes a conflicting transaction are debatable. Most experts would agree with some version of the rules on this slide. The first rule is sound enough, although the condition on reading is very important: we would need to include things like aggregation in our definition of reading. The first rule is also very easy to implement, especially if tables are shareable structures, as a simple 64-bit comparison is sufficient!

For the other rules, we would need to be very clear on what a specified row is, and the non-existence of a row might only be determined by reading the whole table. Such debate is beyond the scope of today's tutorial.

## Demo 2: Commit, Conflict



- ▶ In this demonstration, we will see that during commit of a Transaction  $T$ , we do a validation check
  - ▶ It ensures that the elementary operations of  $T$  can be validly relocated to follow those of any transaction  $T'$  that has committed since the start of  $T$
  - ▶  $T$  planned 
  - ▶ But now 
  - ▶ Relocation amounts to swapping the order of the elementary operations  $e_i$
  - ▶ Two such cannot be swapped if they *conflict*
  - ▶ E.g. They change the same object (write/write conflict)
  - ▶ There are also tests for read/write conflicts between  $T$  and  $T'$

30

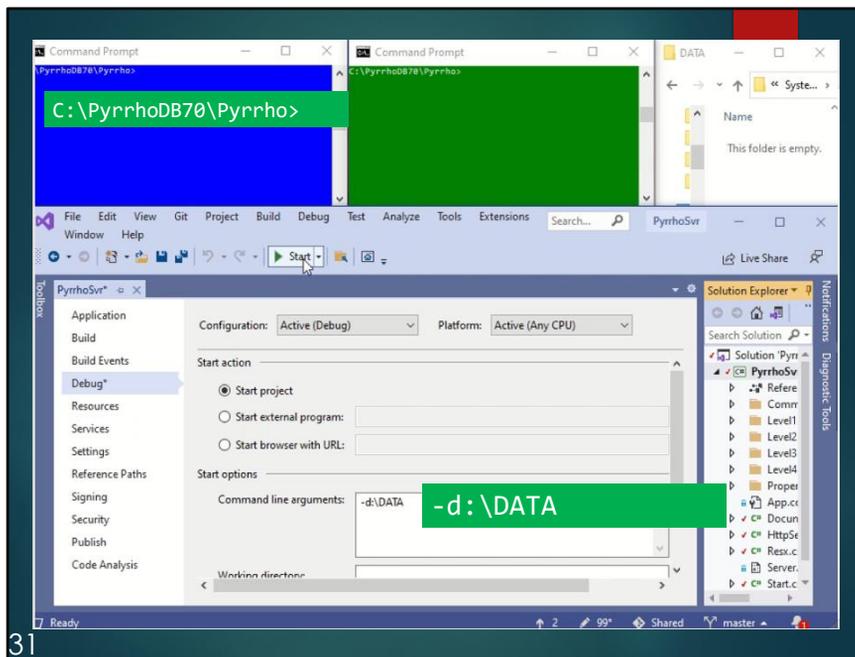
In the second demonstration, we look in detail at the Commit() method for a transaction, and the detection of conflicts.

At the start of Transaction Commit, there is validation check, to ensure that the transaction still fits on the current shared state of the database, that is, that we have no conflict with transaction that committed since our transaction started.

If that is the case, we can relocate all our proposed changes to come after the committed transactions.

The tests for write-write conflicts involve comparing our list of physicals with those of the other transactions.

For checking read-write conflicts, we collect “read constraints” when we are making Cursors.



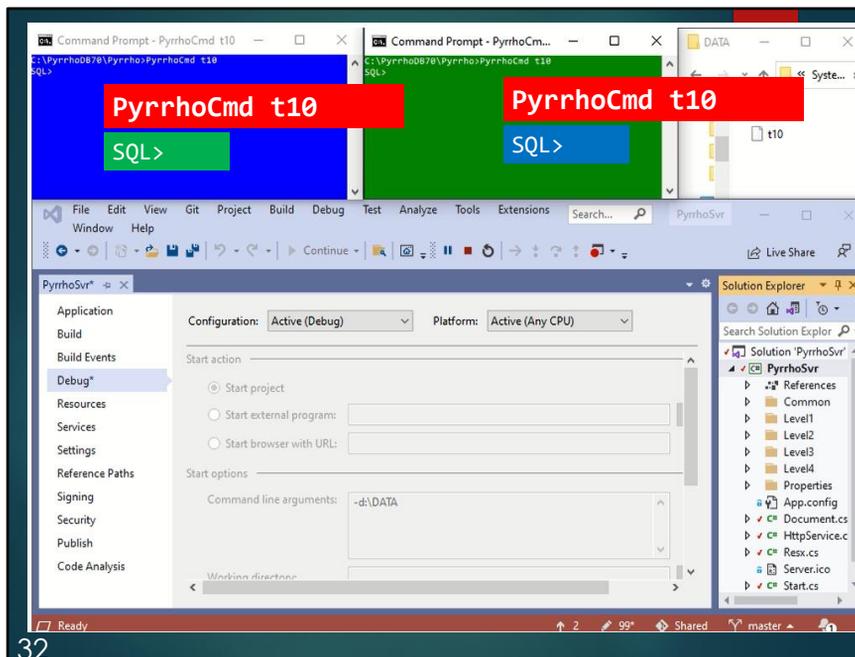
31

The slide shows two client command windows, the database folder, and Visual Studio. We will get the Visual Studio debugger to run the server for us, as this will enable us to look at the validation step of `Transaction.Commit()`.

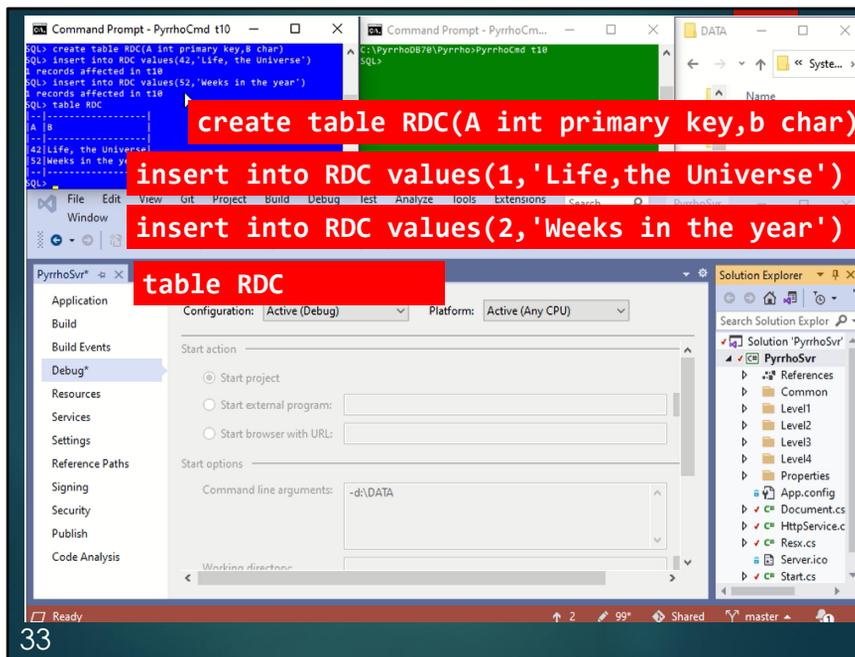
Visual Studio has opened the solution in Pyrrho's `src\Shared` folder, and I have set the database folder to `\DATA` for convenience, using the Debug properties of the `PyrrhoSvr` project. In the command window that the start-up creates, I clicked the Enter key to start the server as in demo 1. I have minimised the server window

to get it out of the way.

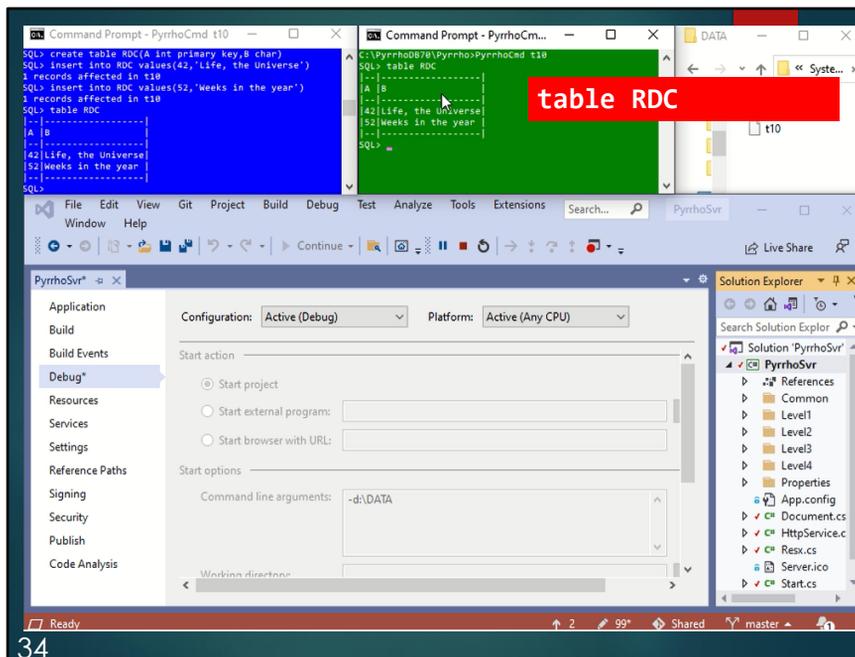
We confirm that the database folder I have nominated is empty.



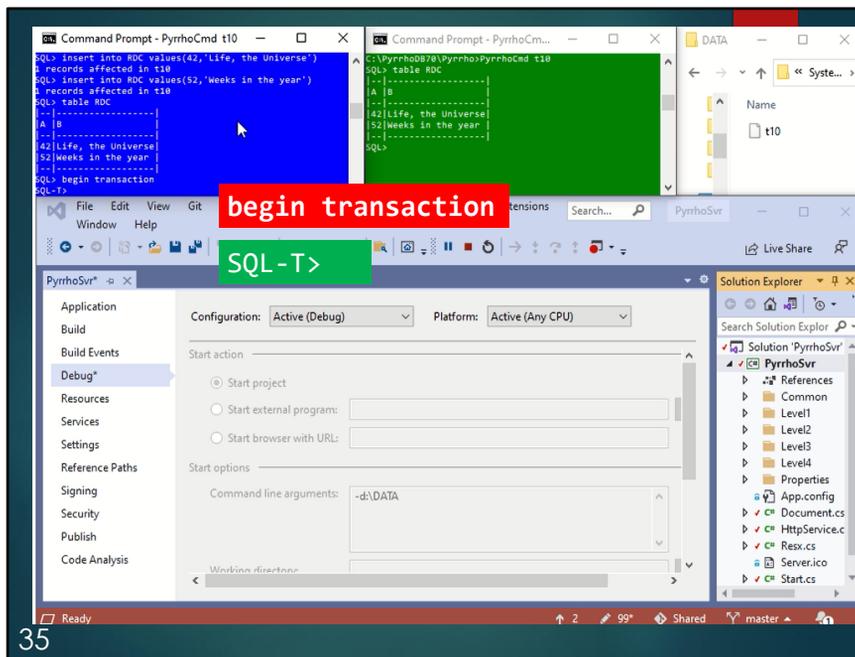
The command we want to run in both windows is the same: PyrrhoCmd t10. Once we do one of them, the DBMS immediately creates the database as we have seen in demo 1 - I have hidden the folder window.



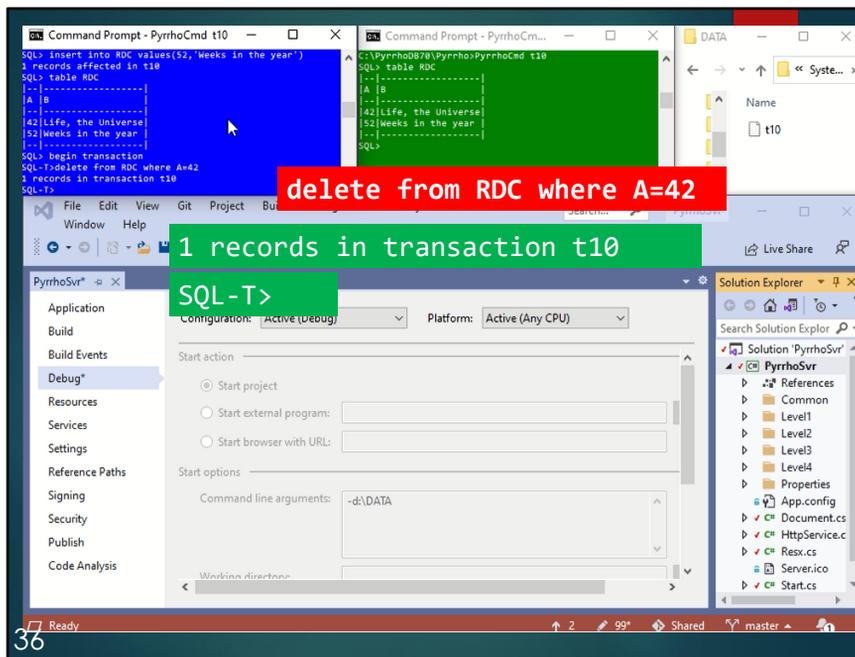
In the blue window we do some work on the database. We create a table and insert a couple of records. We can display the table and see just what we have created there.



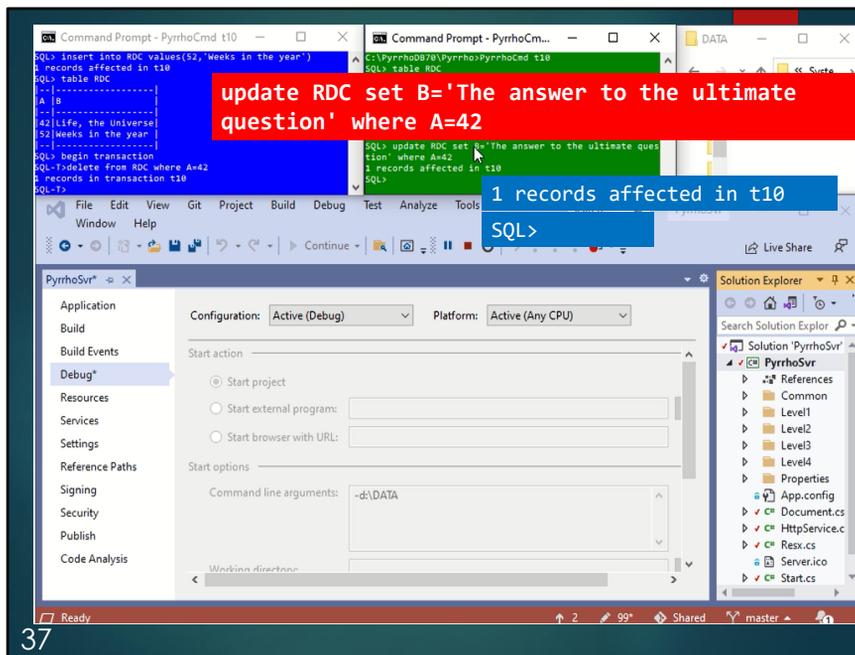
That has all been committed, because we are running in auto-commit mode, so the green window will pick it up. It is also running in auto-commit mode, so a new command starts a new transaction, and it will check the current state of the database. This completes the set up for this demo. We will repeat this part of the demo later.



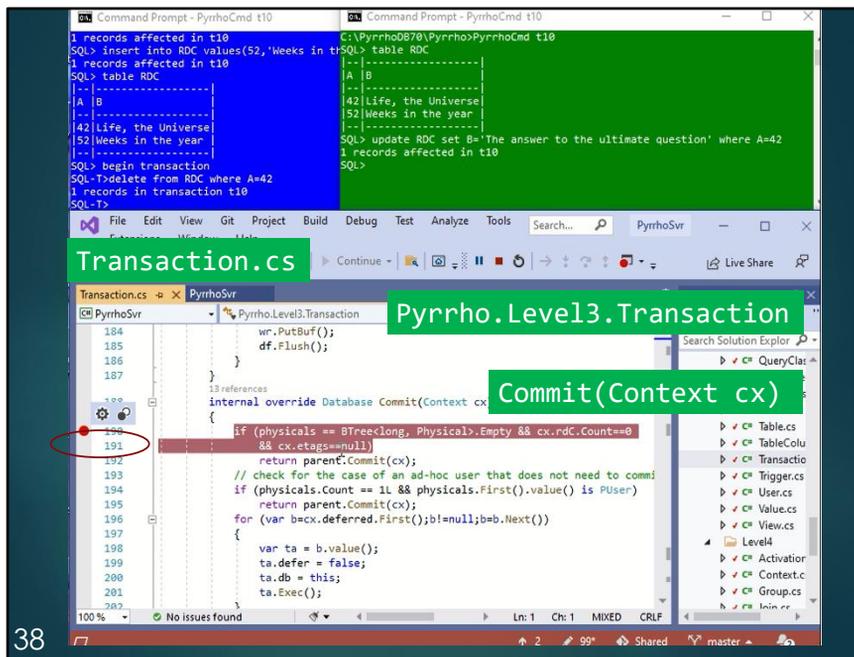
Now we start an explicit transaction in the blue window. If we just relied on auto-commit mode it is very difficult to synchronise an overlap of transactions. For a demo, it works very well to use explicit transactions. Notice that in an explicit transaction the prompt changes to SQL-T>. As long as the transaction is running, we will get these prompts. When the transaction is over, either because of commit, or rollback, or because we have done something wrong, it will go back to SQL>.



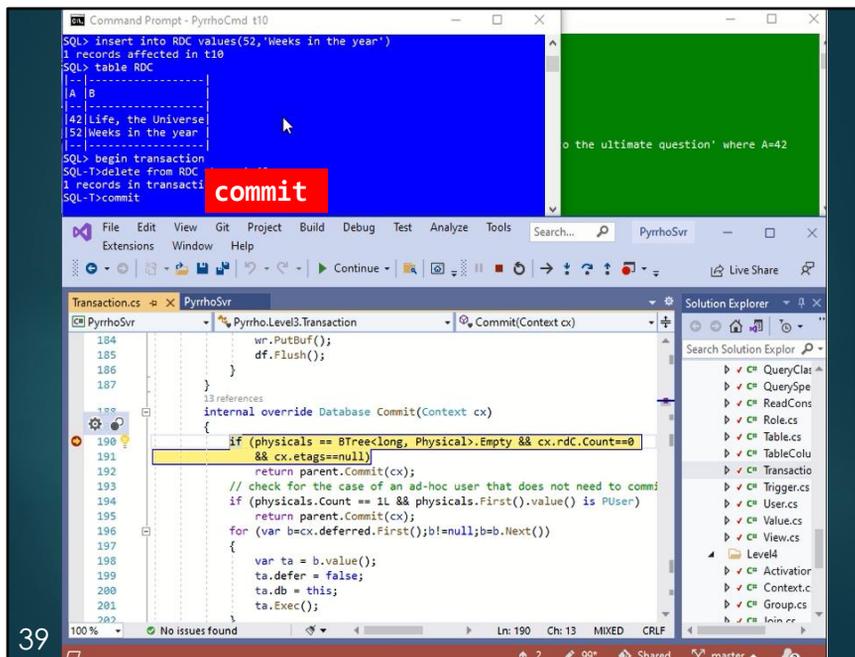
We will request conflicting changes to table RDC in these two clients. In the blue window, let's "delete from RDC where A=42", to delete the first row. That's in a transaction, so nothing has been written to disk yet.



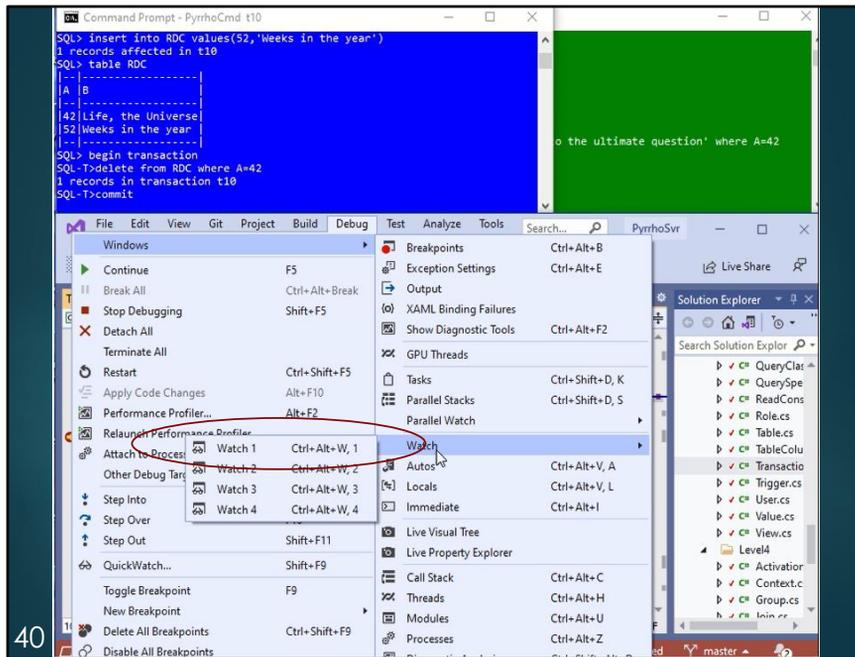
In the green window, we are still in auto-commit mode and we are going to make an update to the same row, which will be committed straightaway to disk. This should make the blue window unable to commit its transaction because of the conflict.



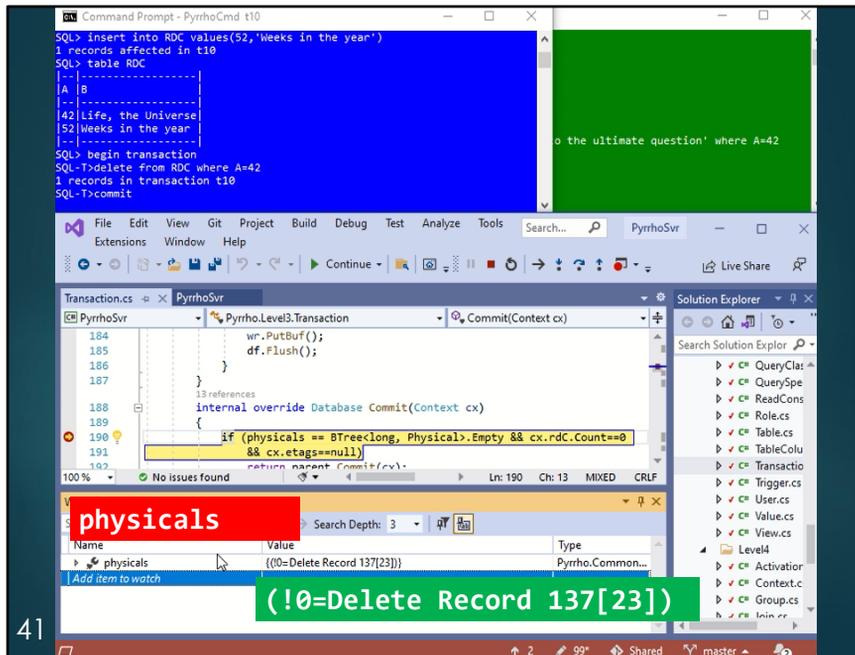
In order to see what happens, let us set a breakpoint in the debugger, at the start of the `Transaction.Commit()` method.



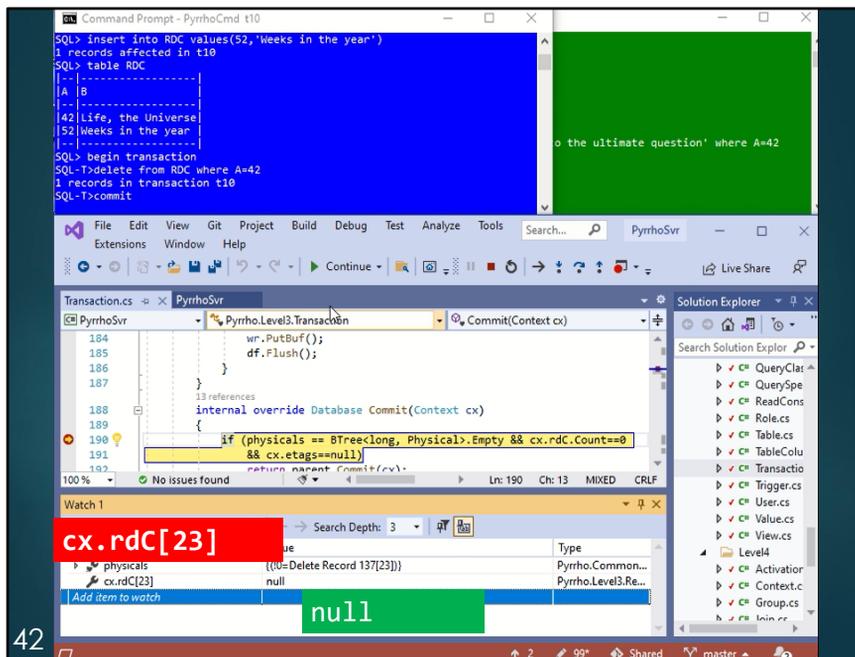
Then when we issue the commit command in the blue window, we hit the breakpoint.



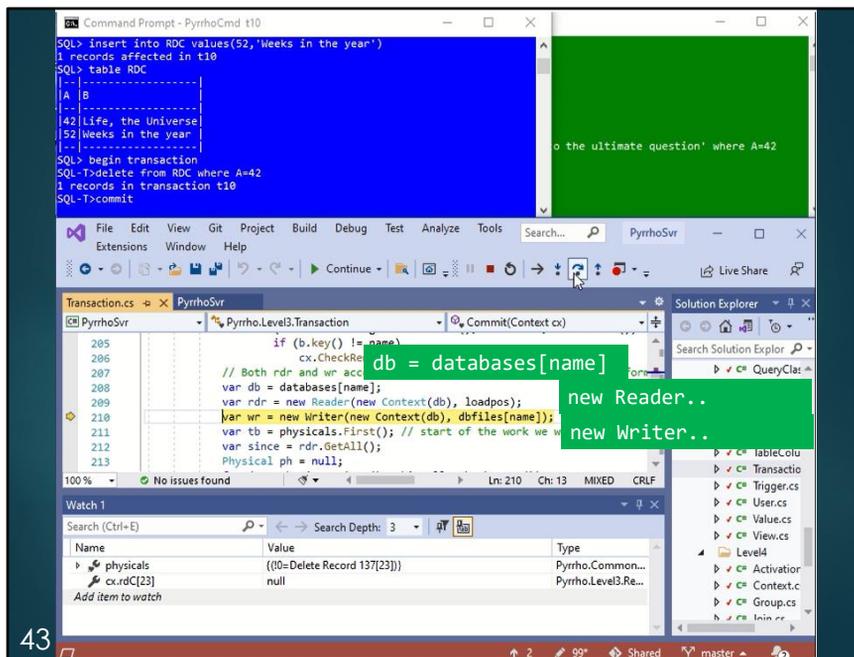
Let us open the Watch window in the debugger, so we can examine details.



Let's look at the `physicals`. `physicals` is the list of Physical records that the Transaction wishes to commit, and it's just the single Physical record to delete row 137 in the database, "Life, the Universe".

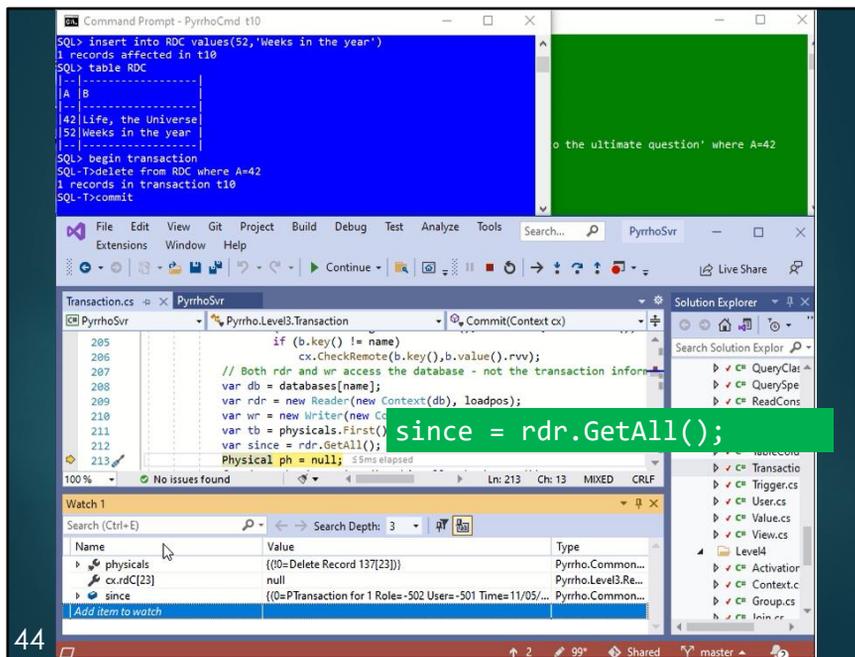


We will also look at `cx.rdC[23]`, which is to do with the ReadConstraints for this particular Transaction. In this case there aren't any, as this transaction hasn't read anything. (In the next experiment we will see it is the other way round.) Step Over a few times to get to line 208.

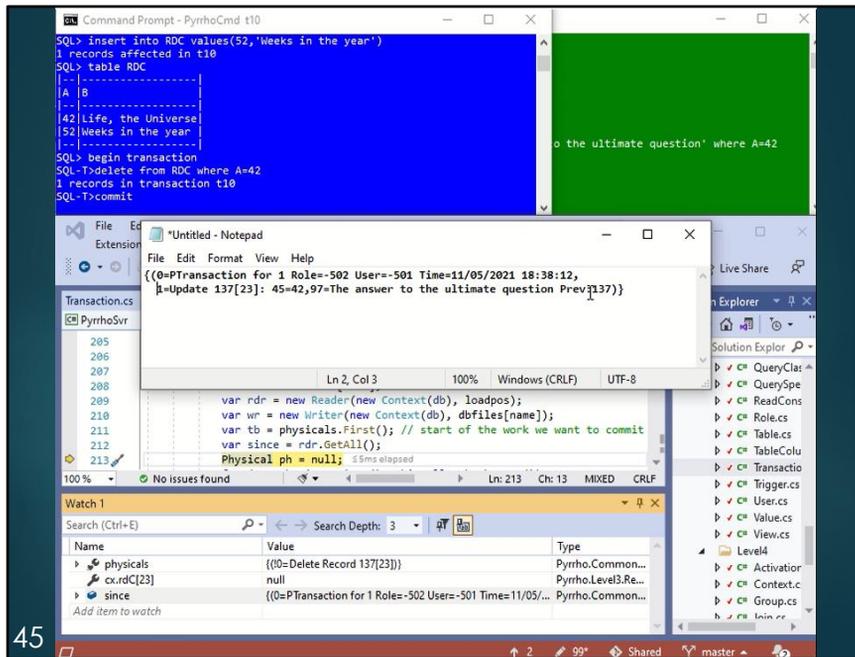


We Step Over a few lines to line 208. The validation step will use the up-to-date copy of the database, as it was left by the green window. We also open a Reader and a Writer: a Reader to look at this database, specifically at the records that have been committed since the start of our Transaction; and a Writer, where we will prepare the records that we are going to add to the database if our validation succeeds.

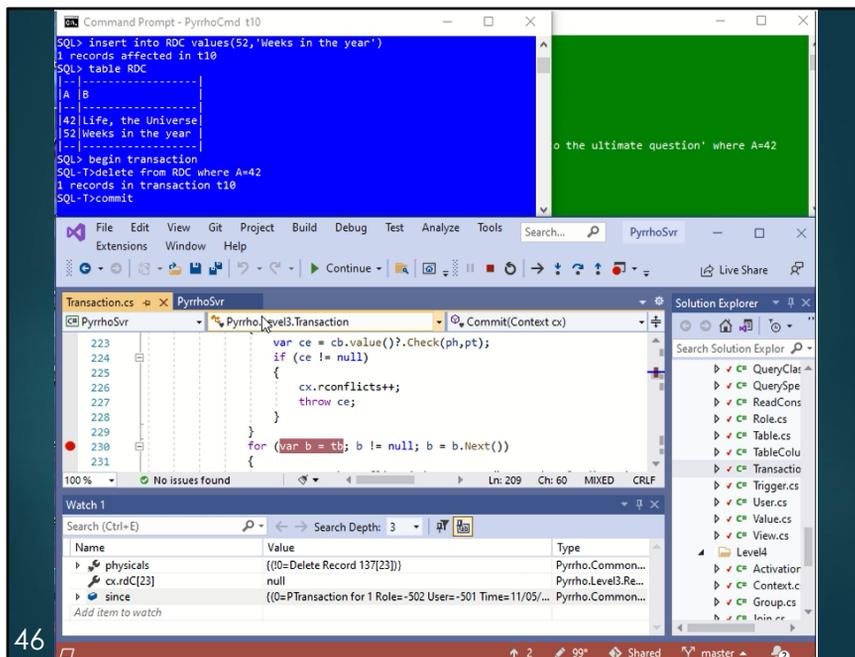
These are not shareable and are subject to locking protocols. They also work on the same FileStream. (Transaction Commit() repeats the validation step after locking the FileStream.)



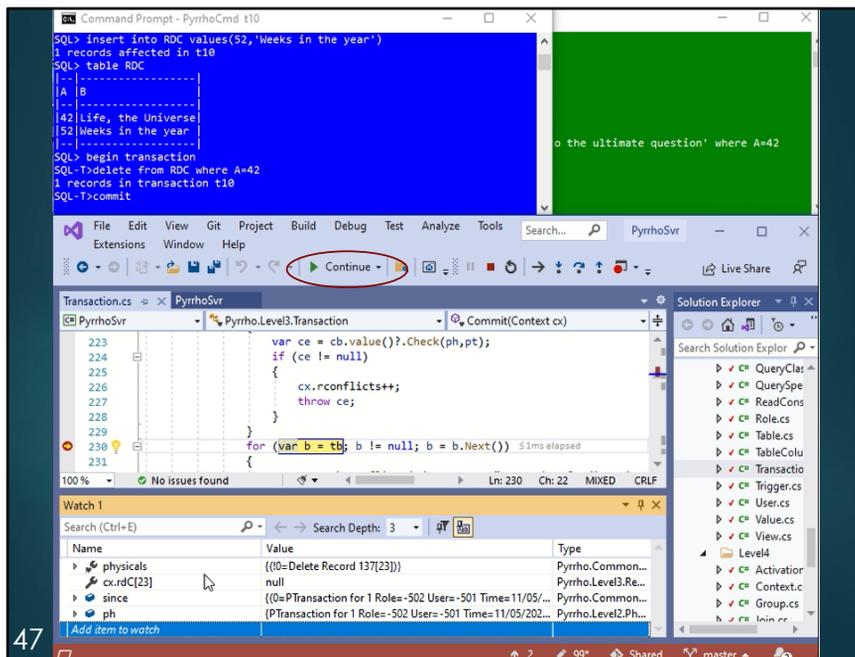
Step Over to line 213, just after a line saying “`since=rdr.GetAll();`” This gets the records that have been committed to the database since the start of our transaction. Use the Watch window to examine **since**. Right-click, Copy Value, and paste it into a Notepad.



We can see we have got the transaction marker and the update that the green window has made.

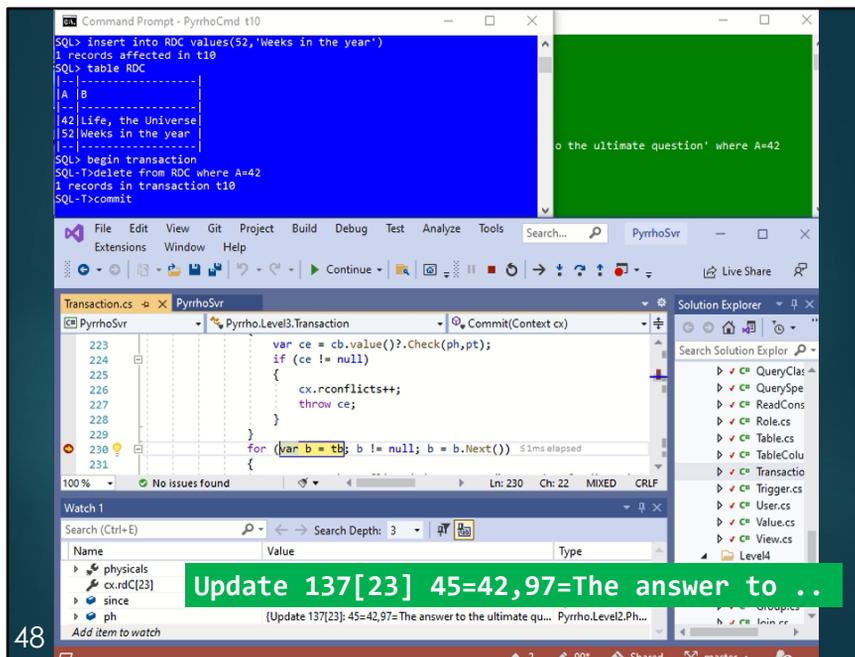


Set a breakpoint at line 230. Click Continue.

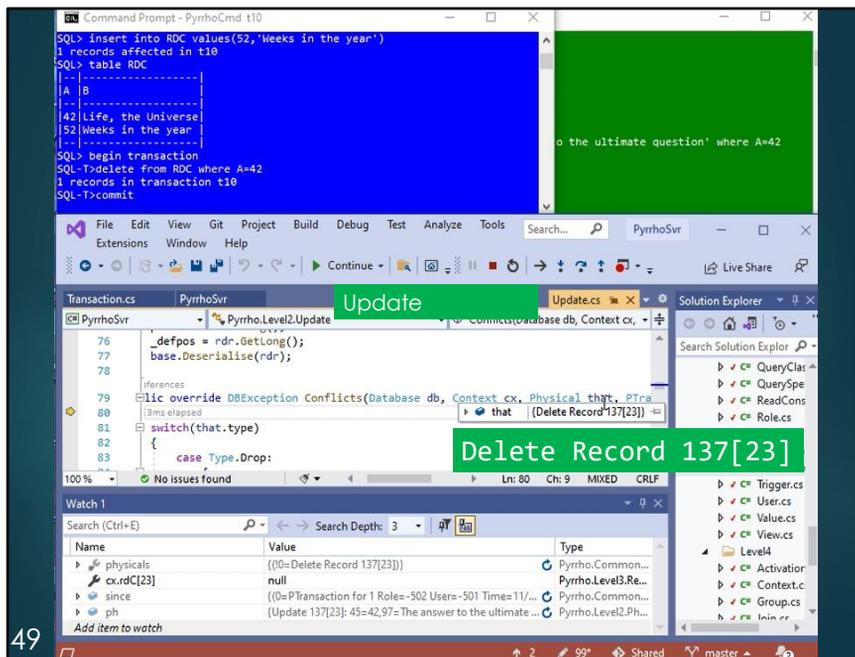


The first time we hit this particular line, ph is the transaction record, which is not very interesting.

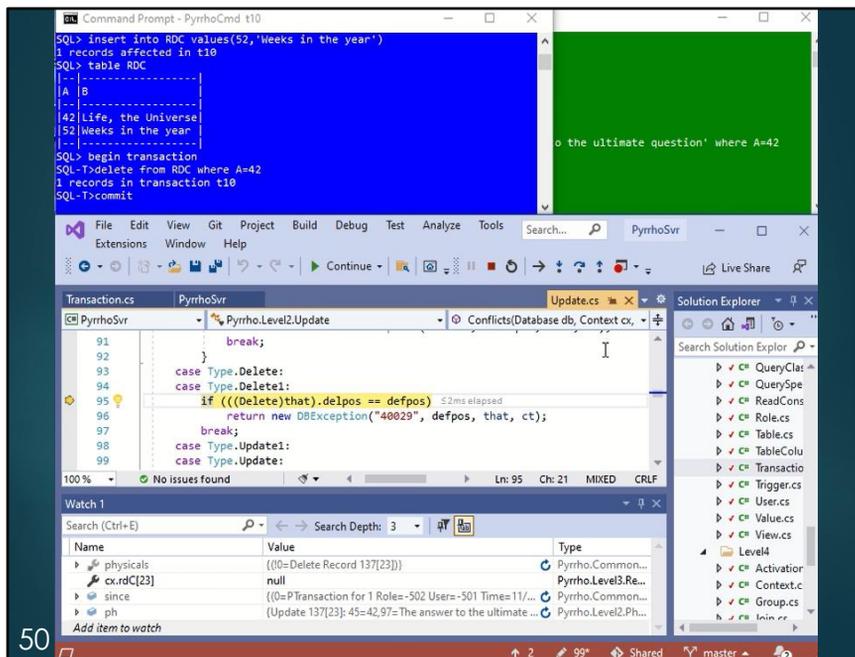
Click Continue



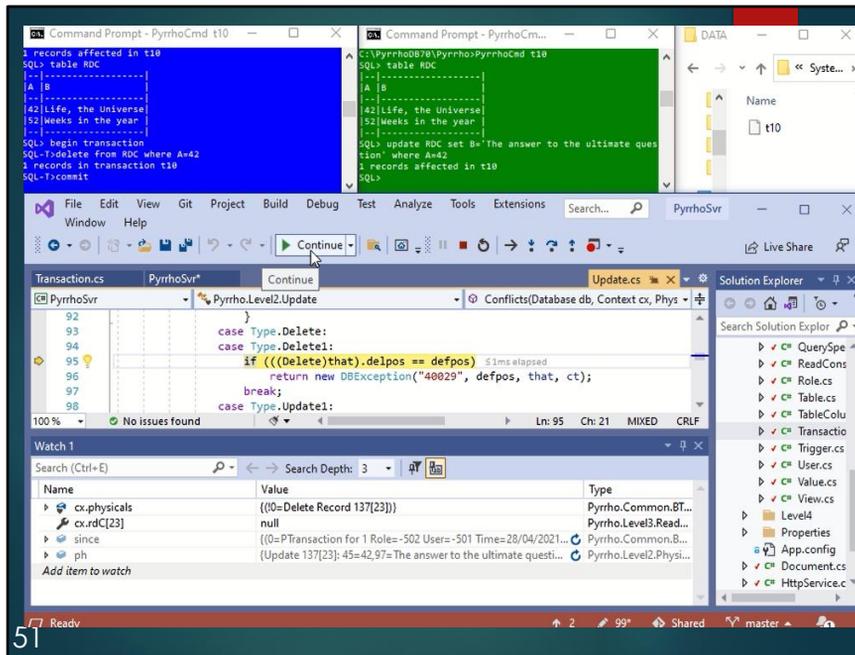
The second time, ph is the Update, and we call the Conflicts() method. Step Into (Visual Studio breaks in evaluation of parameters, so step out, and then in again).



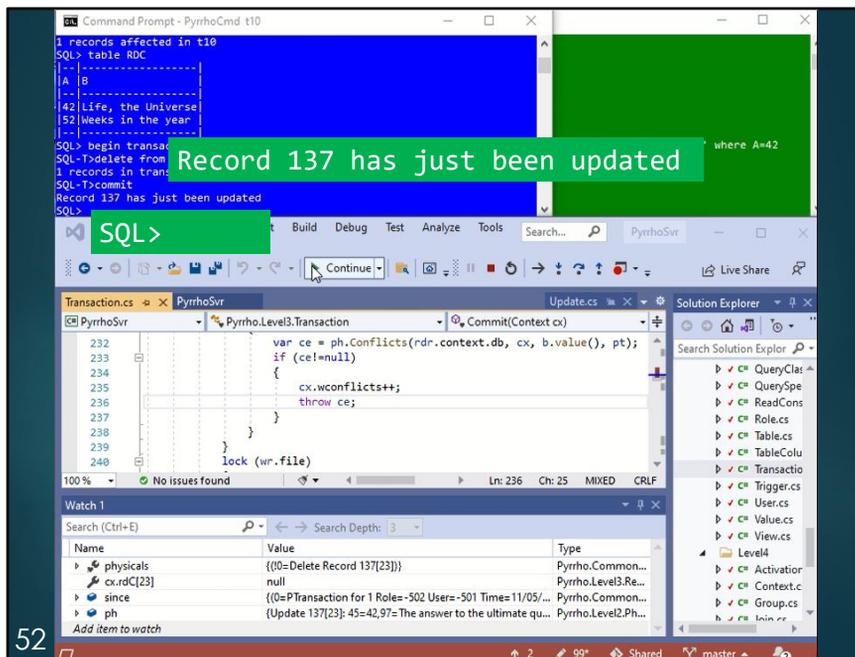
In the Conflicts method, we see that **that** is our Delete, and **this** is the Update. We are looking at the Update that was done by the green window, and comparing it with our Delete record.



Step Over: we reach line 95. For your delete to conflict with an update, it just means that the position you are trying to delete matches the position that has been updated. We can see this is the case, and we will get an Exception. Just click Continue.

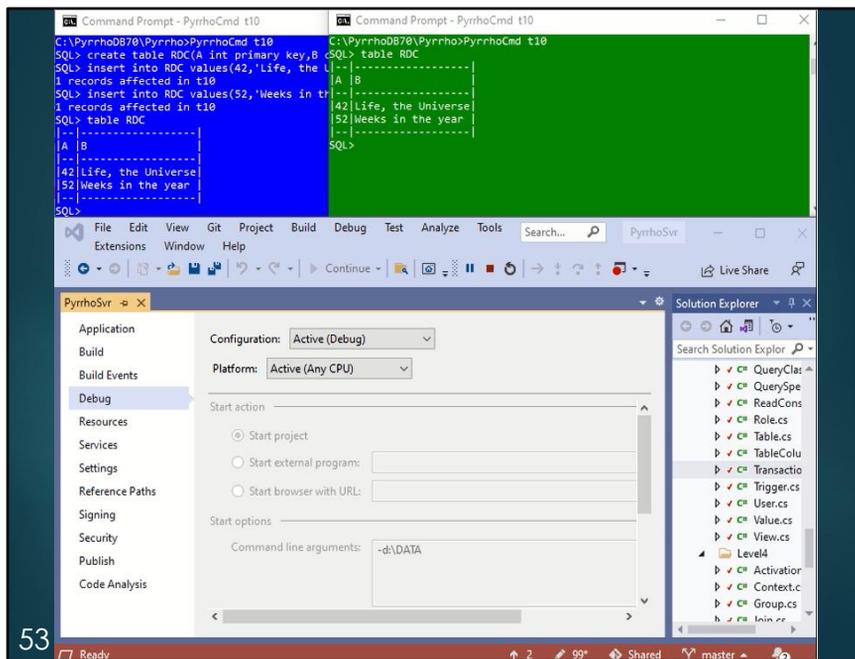


In this case we find there is a conflict, so let's just click Continue



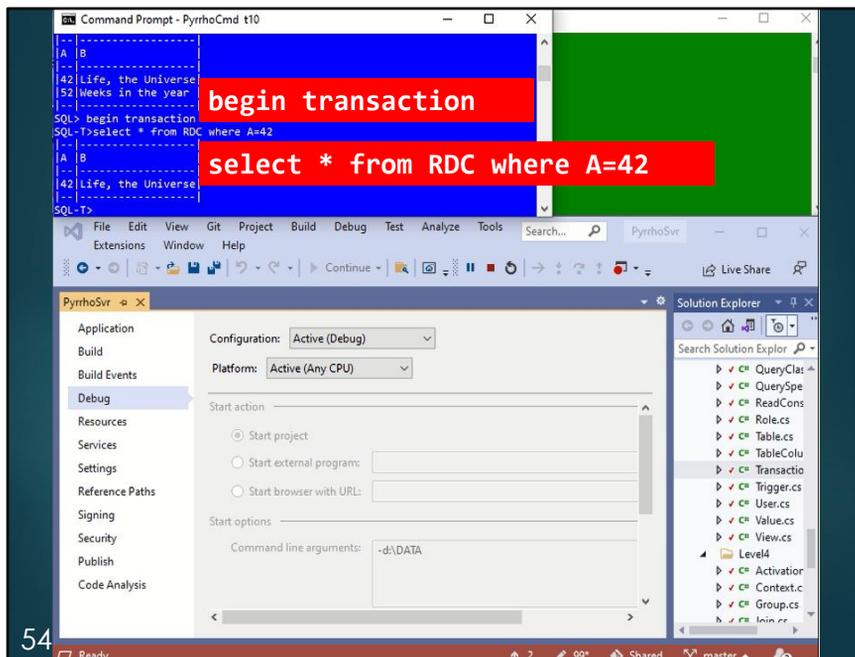
We see that we get a complaint back in the blue window that record 137 has just been updated. The transaction has been rolled back, as we see from the prompt.

That completes the first experiment that we want to do in this demonstration.

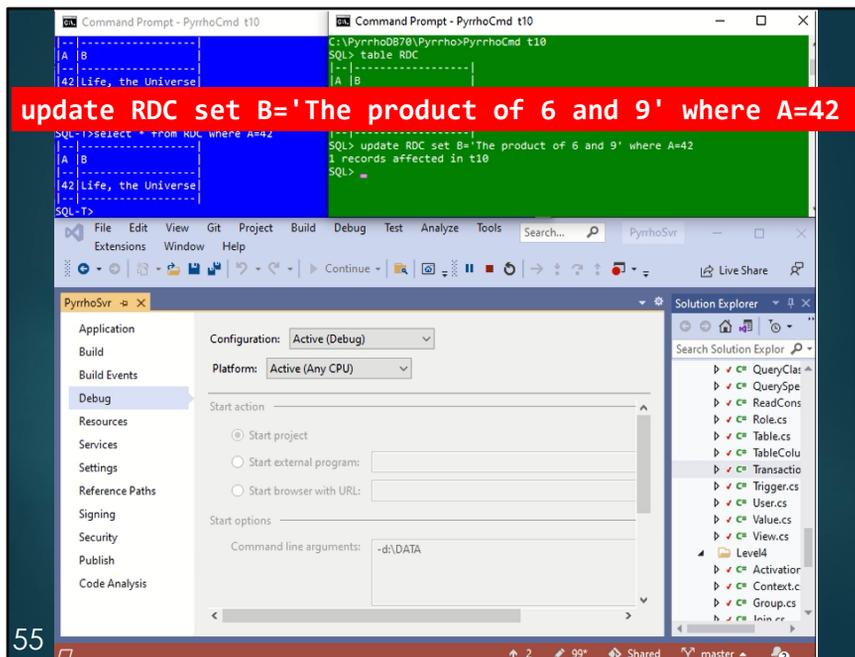


53

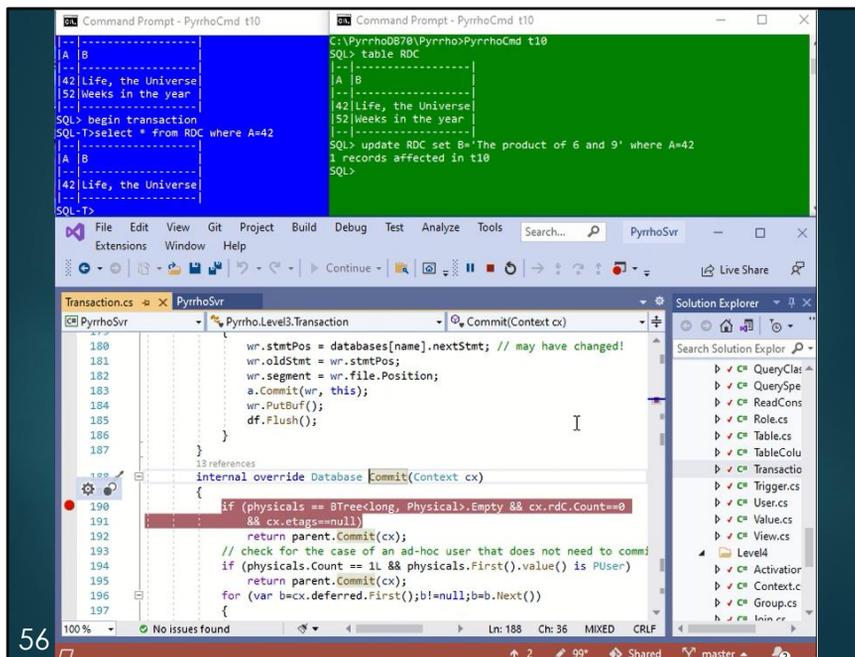
For the second part of the demonstration, for simplicity we will restore the state we had at slide 35: stop the server, delete the database, and repeat the set up to that point.



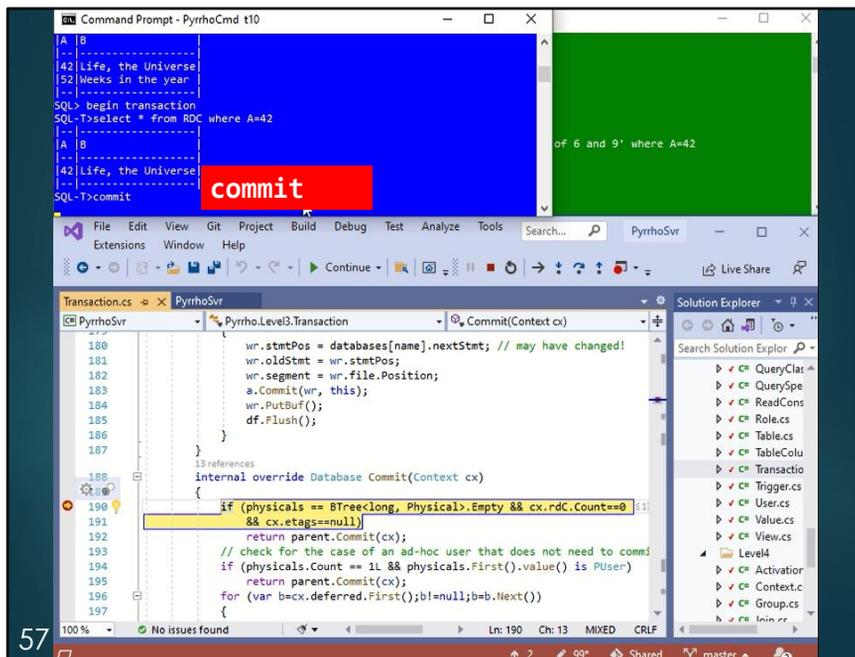
54 This time, when we start an explicit transaction in the blue window, instead of deleting something, we are going to select a single row from the RDC table, and we stick with A=42 again.



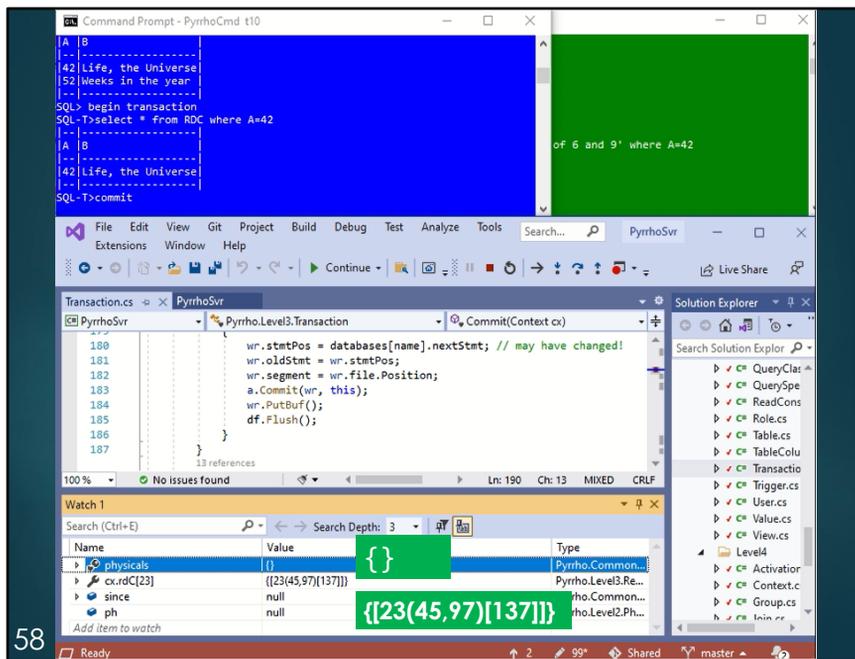
Just as before, the green window makes an update to the same row, which is auto-committed. For the reasons explained earlier, we expect that the blue window will now be unable to commit.



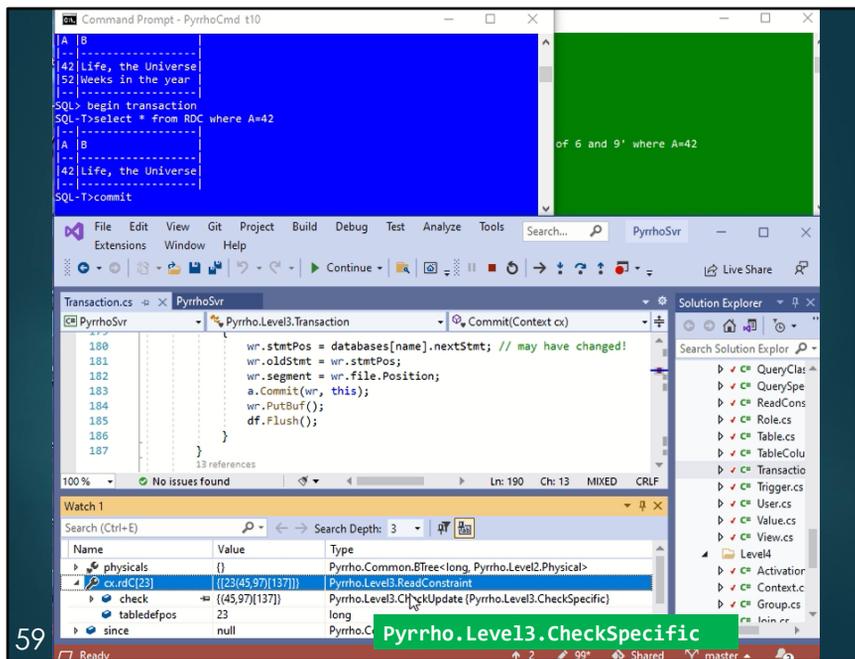
Ensure that we still have a break point in the Transaction.Commit() method.



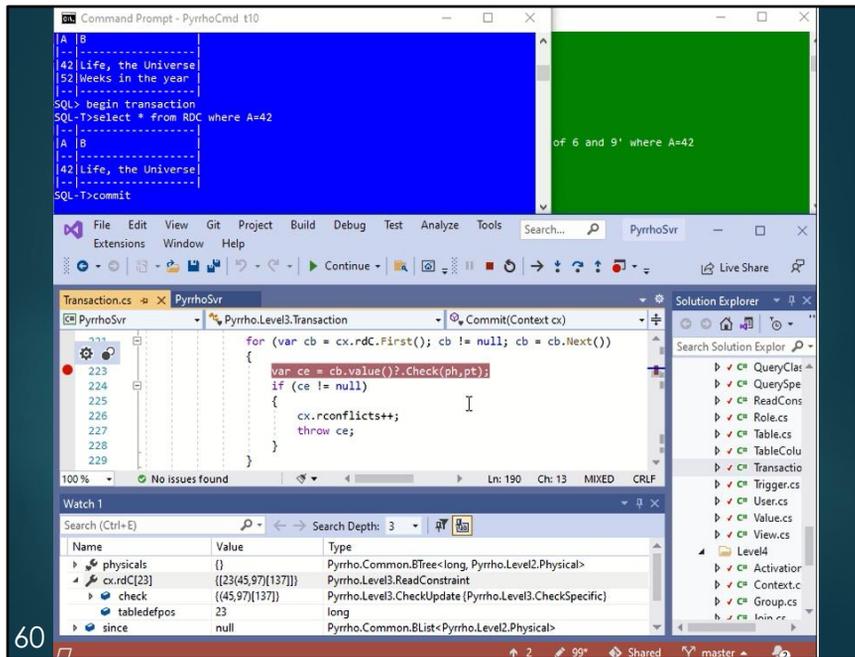
On the commit command, we hit the break point. Open the Watch window again.



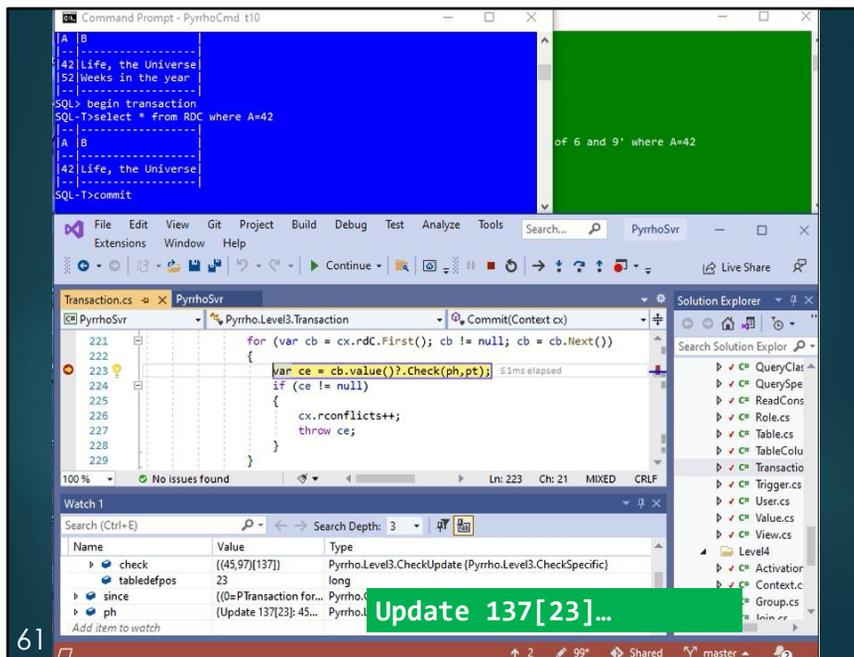
If we look at the Watch window that we had before, we can see that the physicals list is empty this time, but the read constraint has got something in it. (We haven't fetched **since** yet.)



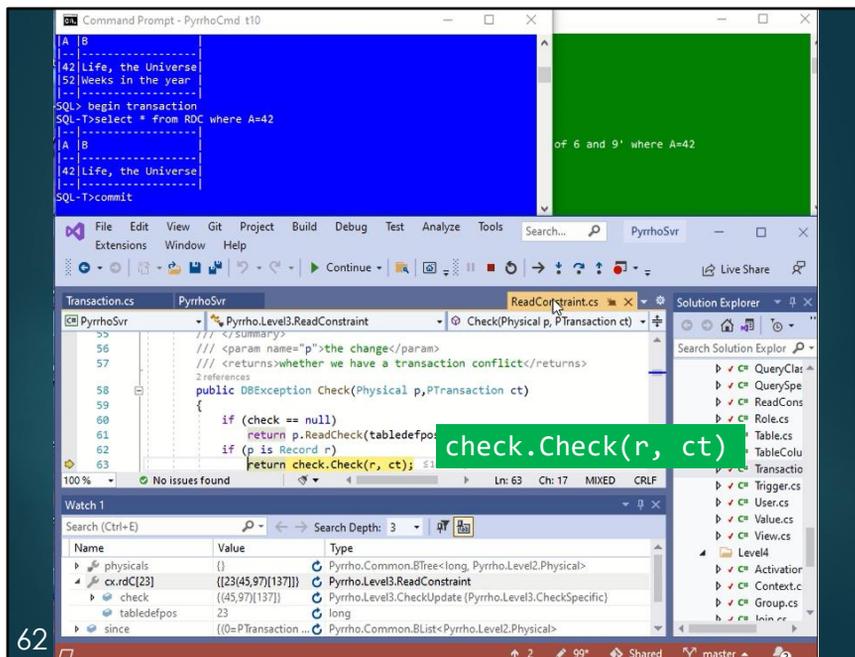
Expand it: we see that this is a CheckSpecific record: the check field shows the columns and the records that have been read.



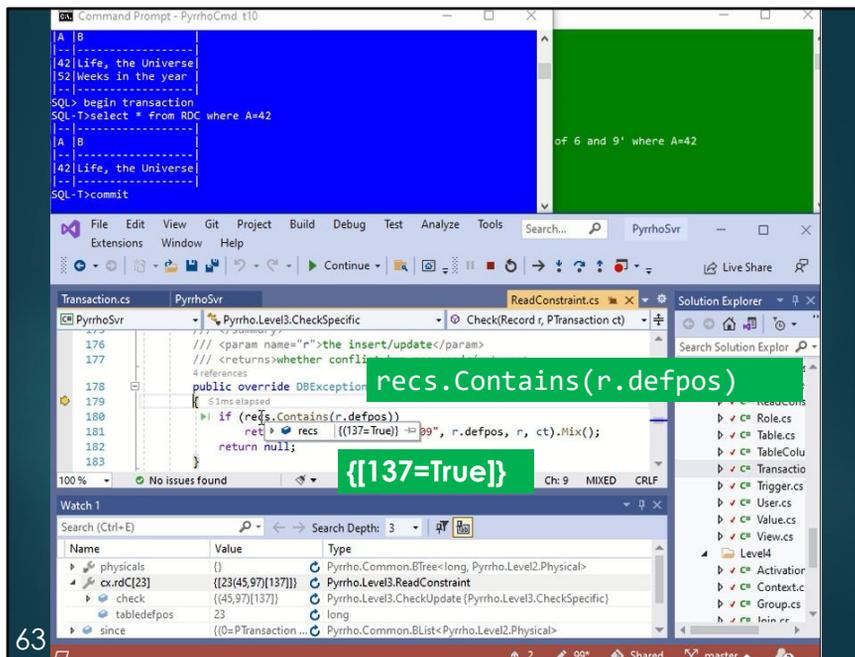
Since rdC is not empty, let's put a break point at line 223



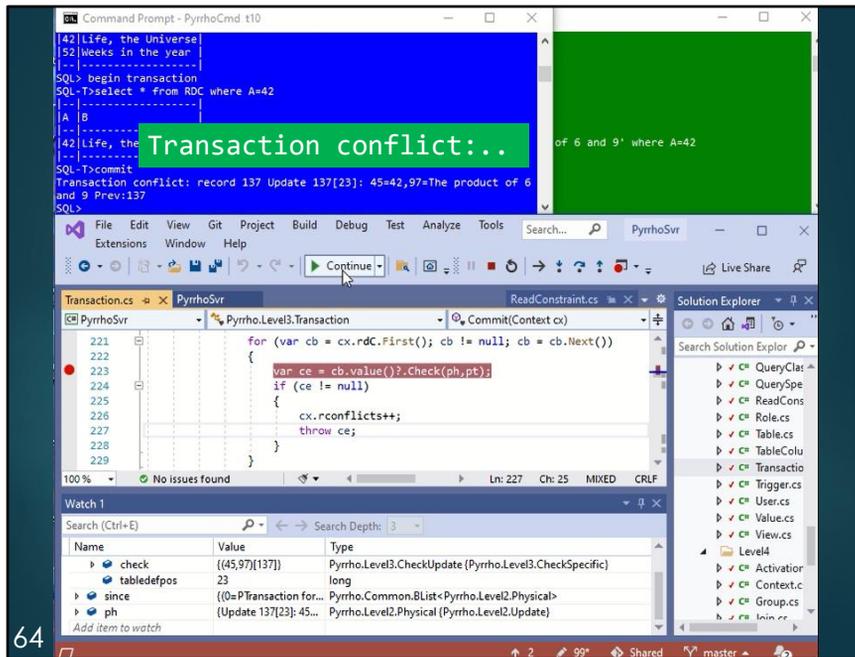
The second time this is hit, ph is the Update that was made by the green window. (The blue window just made a select.) Step Into the Check() call (step out of parameter evaluation, and step in again).



When we step into the ReadConstraint.Check() method, check is the CheckSpecific, and it checks the record r that has been committed by the green window. Step Into check.Check().



This looks at the list of records of the CheckSpecific, the records that were read by the blue window and it contains the record that has just been updated, so an exception occurs. Continue from this point.



A suitable report is sent to the blue client, and the transaction is rolled back. This concludes the demonstration of transaction conflict.

## Shareable structures



- ▶ In a programming language with references
- ▶ Good idea to make all fields **readonly**
- ▶ If the fields are also immutable
  - ▶ Then a reference can be safely shared
  - ▶ So all fields might as well be **public**
- ▶ Also the object can be copied in a single machine instruction `a:=b;` with all fields
  - ▶ No need to clone or deep-copy fields
  - ▶ Assignment, or snapshot, thread safe
- ▶ If we make some changes to the object
- 65 ▶ Other fields can remain safely shared

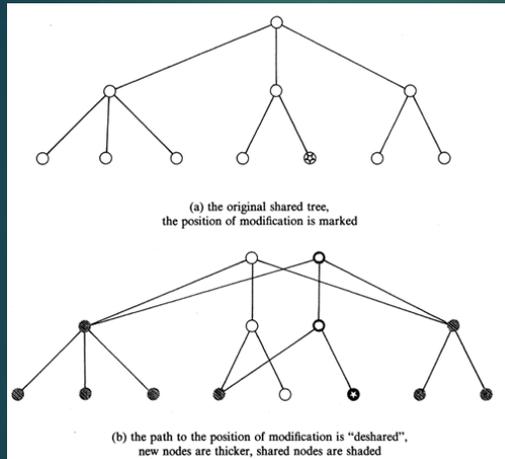
The next question is how best to implement shareable data structures. In a programming language based on references, such as Java, or C#, we can make all fields in our structure final, or readonly. Then any reference can be safely shared, and all fields might as well be public (unless there are confidentiality issues).

If all of the fields are, in turn, also known to be immutable, then there is no need to clone or copy fields: copying a pointer to the structure itself gives read-only access to the whole thing. For example, if the **Database** class is shareable, and `b` is a database, then `a:=b` is a

shareable copy of the whole database (we have just copied a single 64-bit pointer). This is also guaranteed to be thread-safe.

Pointers to shareable structures are never updated, but can be replaced when we have a new version to reference. If this new version has some changed fields, it is perfectly fine to continue to use the same pointers for all the unchanged fields.

## When we add a node



66 [Krijnen and Meertens, 1982]

When we add a field located deep in a shareable structure (e.g. a node to a shareable tree structure), we will need to construct a single new node at each level back to the top of the structure. But the magic is that all the other nodes remain shared between the old and new versions.

The picture shows a tree of size 7, and updating (that is, replacing) one leaf node has resulted in just 2 new nodes being added to the tree. This makes shareable B-Trees into extremely efficient storage structures.

In tests, we see that for a suitable minimum number of

child nodes in the B-Tree, the number of new nodes required for a single update to a B-Tree of size  $N$  is  $O(\log N)$ , and experimentally, this means that for each tenfold increase in  $N$ , the number of new nodes per operation roughly doubles.

Note that we also get a new root node every time (this avoids wearing out flash memory).

## Why C#?



- ▶ It is helpful if the programming language supports:
  - ▶ `readonly` directives (Java has `final`)
  - ▶ Generics (Java has these)
  - ▶ Customizing operators such as `+=`
    - ▶ Because `a+=b` is safer than `Add(a,b)`
      - ▶ Easy to forget to use the return value `a=Add(a,b)`
    - ▶ Implies strong static typing (so not Java)
      - ▶ Many languages have "type erasure" ☹
- 67▶ Also useful to have all references nullable

Let us choose a popular programming language to recommend. We have `readonly` or `final` directives in C# and Java, and generics in both.

But C# allows us to create statically-defined operators such as `+=`, and this is a great advantage because `a+=b` allows no mistake, while it is very easy to think `Add(a,b)` adds `b` to `a`. (It does, but only if the result of the method is then assigned to `a`.)

Java also lacks strong static typing because of type erasure, and C# allows all references to be nullable by

default.

So I prefer C#, which now has been around for 19 years. Java and Python have been with us for over 30 years. However, C# provides no syntax for requiring a class to be shareable.

## Shareable objects in DBMS



- ▶ Let's make structures in the DBMS shareable
  - ▶ Database, Transaction
  - ▶ Table, Index, TableColumn, Procedure, Domain, Trigger, Check, View, Role
  - ▶ Query, Executable, RowSet, most Cursor
  - ▶ TypedValue
- ▶ Some classes are NOT shareable:
  - ▶ Use mutable `Context` and `Activation` variables
  - ▶ Files and `HttpRequest` are mutable
  - ▶ `Physical` objects are for preparing log records
    - ▶ So cursors that examine logs are not shareable

68

What data structures in the DBMS can be made shareable?

Database itself, and its subclass, Transaction.

Database Objects such as Table, Index, TableColumn, Procedure, Domain, Trigger, Check, View, Role

Processing objects such as Query, Executable, RowSet, and their many subclasses;

Cursor and most of its subclasses.

TypedValue and all its subclasses

All of these can be made shareable.

Context and Activation cannot be made shareable because in processing expressions we so often have intermediate values.

Also, something needs to access system non-shareable structures such as FileStreams, HttpRequest.

And Physical and its subclasses are used for preparing objects for the database file, so cursors that examine logs are not shareable.

## BTree<K,V> and BList<V>



- ▶ BTree is a sort of unbalanced B tree
  - ▶ += adds (key,value) pair, -= removes a key
- ▶ BList is subscriptable subclass where K is int
  - ▶ But is slower: renumbers on insertion and deletion
- ▶ Pyrrho's implementation of BTree is weird
  - ▶ Would be a nightmare to prove correct
    - ▶ But it can be replaced by a better one!
- ▶ Traverse up and down using shareable helpers

```
69 ABookmark<K,V> First(), Last();
```

We have already mentioned BTrees. These are used throughout the implementation and have proved very useful. As mentioned above, BTree<K,V> appears to exhibit logarithmic behaviour, It is also used for implementation of BList<V>, but BList is slower in order to ensure that the keys are always 0,1,.. .

If someone can come up with a better (more provably correct) version of BTree, that would be great!

As described in the StrongDBMS paper, traversal of both BTree and BList is done using shareable bookmarks, and since last year both allow bidirectional traversal.

## ABookmark<K,V>



- ▶ Immutable and shareable

```
internal K key();
```

```
internal V value();
```

- ▶ Continue to traverse the tree they start in

```
internal ABookmark<K,V> Next()
```

```
internal ABookmark<K,V> Previous()
```

- ▶ Obviously they don't see any changes to it
- ▶ Base class for all kinds of Cursors
- ▶ BTree<K,V> is shareable if K and V are

70

This slide give some more detail on Bookmarks. Given a Bookmark we can move to the Next, or Previous entry in the tree or list (and we get a new Bookmark of course). They continue to traverse the tree that returned them (which remains immutable). The Cursor class is a subclass of ABookmark and is the base class for numerous Cursor classes.

These structures are shareable provided that K and V are shareable.

## Basis



- ▶ Base class for objects with properties  
`internal readonly BTree<long,object> mem;`
  - ▶ The key is used to label properties  
`internal const long Name = -50;`
  - ▶ Operator `+` helps with modifications (via `+=`)
  - ▶ Just one `readonly` property  
`public string name => (string)mem[Name];`
  - ▶ Allows us to define shareable properties
  - ▶ Basis will be shareable provided all property values are shareable
- 71 ▶ e.g. `long`, `string`, `DateTime`, or a shareable class

A key building block in the implementation is the Basis class, which is the parent of all of the shareable classes mentioned on slide 69, except TypedValue.

Basis itself contains just one field `BTree<long,object>` called `mem`, and has just one shareable property **name**. Object is not shareable: but a Basis object will be shareable if all its property values are.

Its importance comes from the fact that negative keys in `mem` provide a set of readonly properties. Each subclass of Basis can define its own property keys.

So `Name` is `-50`, and negative keys from `-50` to `-500` are reserved for such properties.

Keys below `-500` are used for predefined types and system tables.

Any subclass of Basis can define `operator+` to give useful ways of updating properties.

## Basis, contd



- ▶ Basis has an abstract method

```
abstract Basis New(BTree<long,object> m)
```

- ▶ So a subclass such as Framing must have

```
public override Basis New(BTree<long,object> m)
{ return new Framing(m); }
```

- ▶ Should also have a way of changing its properties

```
public static Framing operator+(Framing f,(long,object) x)
{ return (Framing)New(mem + x); }
```

- ▶ Here in mem+x we have the operator+ from slide 69

- ▶ Example: if d is declared as Database d;

- ▶ Then we can have d += (Name, "Fred");

72 ▶ += for a Database returns a Database (covariance)

This slide adds some more technical detail for the Basis class and how its subclasses work.

Every subclass must implement New. Provided every subclass also implements + similarly to the one shown, we do not need a new constructor for modifying every kind of field in a shareable subclass.

Then whatever subclass x is declared to be, x+(k,v) will be the same class for any long k.

## DBObject



- ▶ DBObject is a subclass of Basis
  - ▶ has a `readonly long` field called `defpos`
  - ▶ Otherwise uses the Basis machinery for properties
- ▶ The `defpos` is a unique identifier (uid) for every object in the database
- ▶ There are different ranges, e.g.
  - ▶ negative for system objects, properties etc
    - ▶ -1L is used for an undefined property
  - ▶ objects such as tables have a defining position in the transaction log
  - ▶ Ranges for objects with shorter lifetimes, e.g. heap, lexical positions, uncommitted objects

73

DBObject is a very useful abstract subclass of Basis. It has one new field `defpos`, which is a `readonly long`. This is where unique identifiers come in: all `defpos` are uids. The range of values `long.MinValue` to `long.MaxValue` is divided into 6 ranges as described here. Uids are faster to compare than string identifiers, and easy to come up with a next value.

## Database

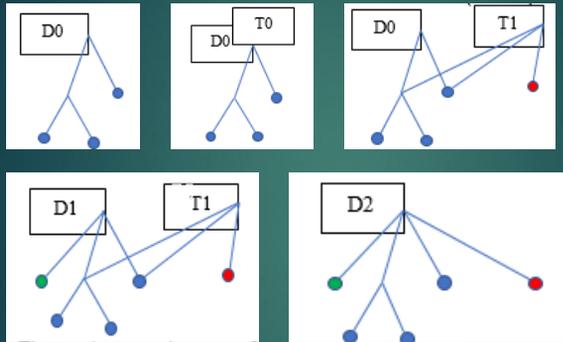


- ▶ Database is a subclass of Basis
- ▶ Lots of properties, and uses mem for objects  
`public BTree<long,object> objects => mem;`
- ▶ Two important static lists  
`protected static BTree<string, FileStream> dbfiles;`  
`internal static BTree<string, Database> databases;`
- ▶ Database has a `Load()` method to build from the transaction log on startup
- ▶ Transaction is a subclass of Database
  - ▶ With a `Commit` method to append its actions to the database file

74

In the demos we have seen reference to databases and dbfiles. Here we see how they fit into the overall design.

## Transaction and B-Tree



75

The StrongDBMS paper gives a step-by-step account of the way Transaction commit works. This is basically a BTree implementation for the process described earlier (slide 27).

Suppose we have a database in any starting state  $D_0$ .

If we start a transaction  $T_0$  from this state, initially the  $T_0$  is a copy of  $D_0$  (i.e. equal pointers).

As  $T_0$  is modified it becomes  $T_1$ , which shares many nodes with  $D_0$ , but not all.

$D_0$  also evolves as some other transaction has committed, so  $D_1$  has a new root and some new nodes.

When  $T_1$  commits, we get a new database  $D_2$  incorporating the latest versions of everything.

## RowSet review



- ▶ Parser turns SQL statements into Executables
- ▶ Execution is on RowSet not Query objects!
- ▶ So in v7 we review rowsets at end of parse
  - ▶ No point in optimising queries
- ▶ SELECT <Items> FROM <TableExpression>
  - ▶ Many RowSet types work on simple columns
  - ▶ Others allow expressions for items, tables
- ▶ Many RowSets are updatable
  - ▶ Some more with help of adapter functions...

76

## Many other DBMS describe their process of “Query optimisation”.

The result of parsing data manipulation language in Pyrrho v7 is not an optimised Query but a possibly updatable RowSet.

So instead of Query Optimisation, Pyrrho v7 has RowSet review. In this process as we will see, filters and aggregations are pushed down from the query level as far as possible towards the table level.

For example, during Rowset Review we replace selection from views and joins by selection underlying tables. This leads to a simple implementation of updatable views and joins.

## Demo 3 : Views



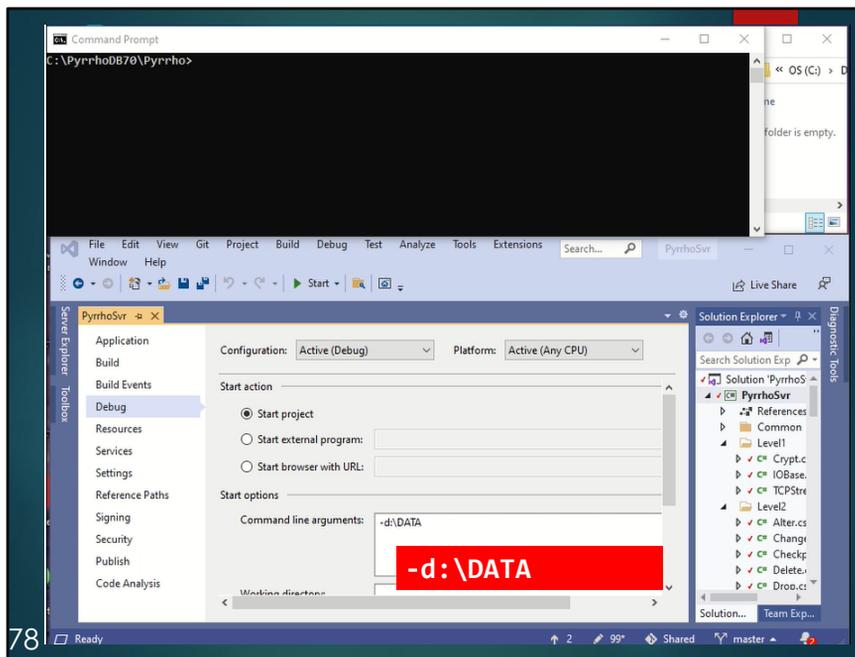
- ▶ This demonstration explores the operation of RowSets
  - ▶ We will see that some RowSets can be used for update, insert and delete actions in addition to queries
  - ▶ We will see the use of precompilation of complex database objects
  - ▶ We will see that RowSet analysis helps ensure that operations on Views are implemented as operations on the (possibly remote) base tables
- 77 ▶ But only if the user has the right permissions

This demonstration explores the operation of RowSets

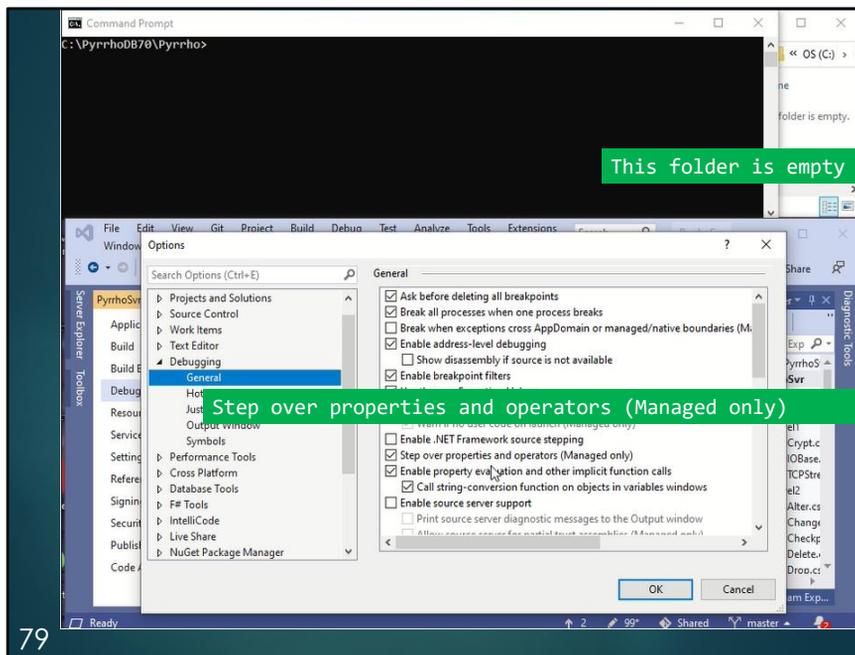
We will see that RowSets can be used for update, insert and delete actions in addition to queries

We will see the use of precompilation of complex database objects

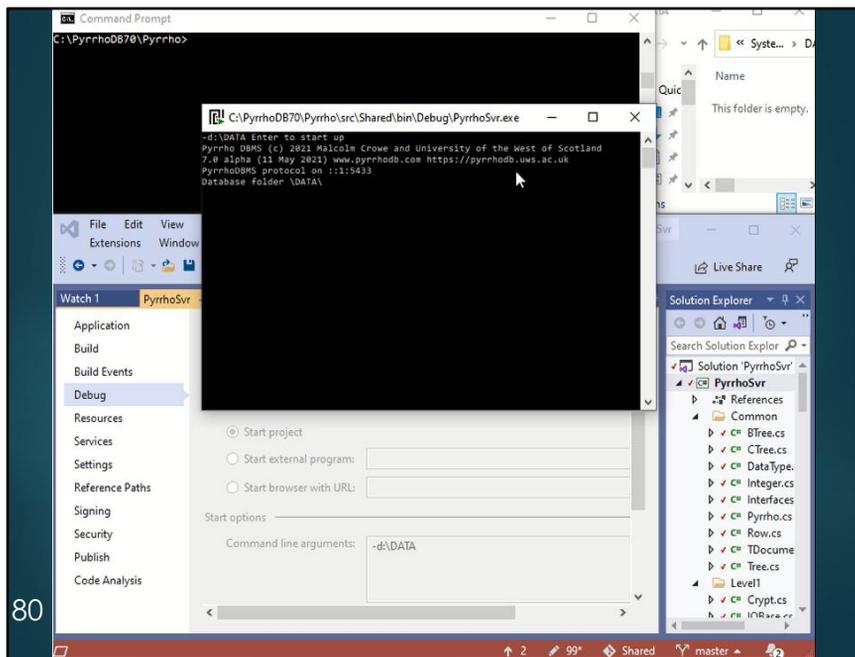
We will see that RowSet analysis helps ensure that operations on Views are implemented as operations on the (possibly remote) base tables, but only if the user has the right permissions.



The slide shows a client command window, a glimpse of a File explorer, and a window for Visual Studio. This demonstration traces through part of test12 of the PyrrhoTest program using the Visual Studio debugger. In Visual Studio, open the PyrrhoSvr solution in the src\Shared folder of the distribution. Set the debug properties of the PyrrhoSvr project to have -d:\DATA in the Command line arguments.

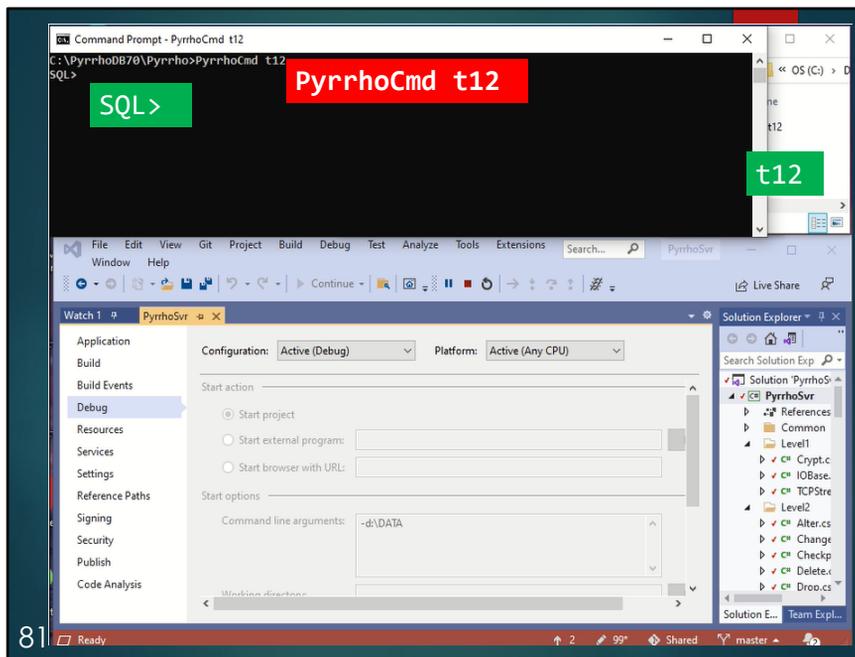


From the Debug menu, select Options.. and ensure that the “Step over properties and operators (Managed only)” in Debugging/General is checked. Click OK.

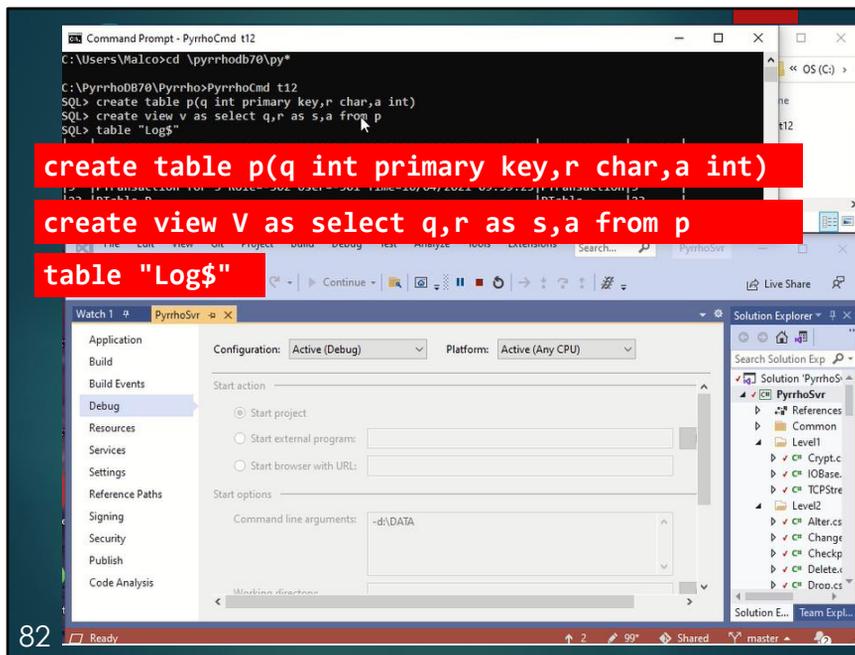


Now click Start in the debugger, and in the popup command window, click Enter. We hide this window because it is not going to do anything interesting during this demo.

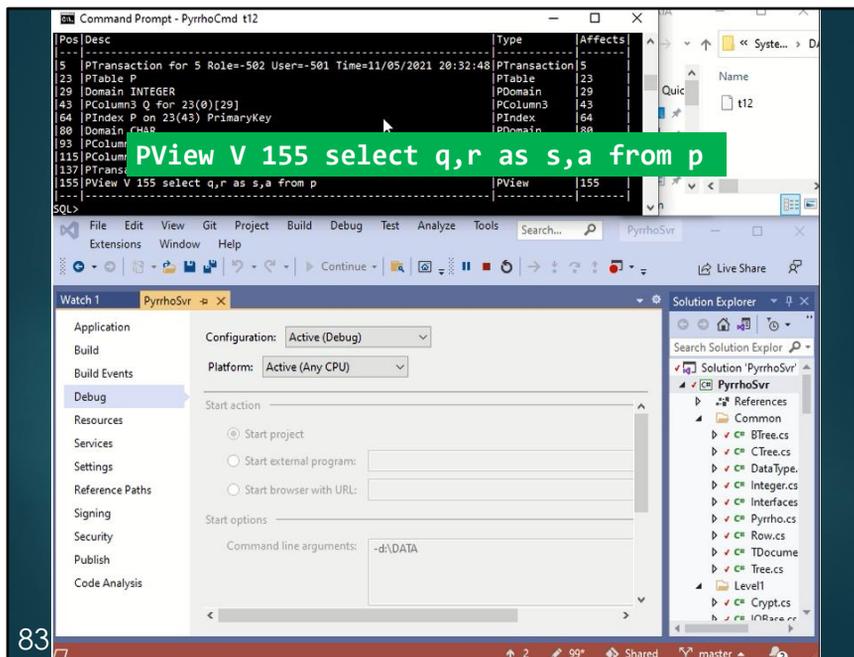
We want to ensure that the database folder that we are using for our databases is currently empty before we start the demonstration.



In a command window set to the distribution folder, start the command line client PyrrhoCmd for database t12. We see that the database is created by the server.

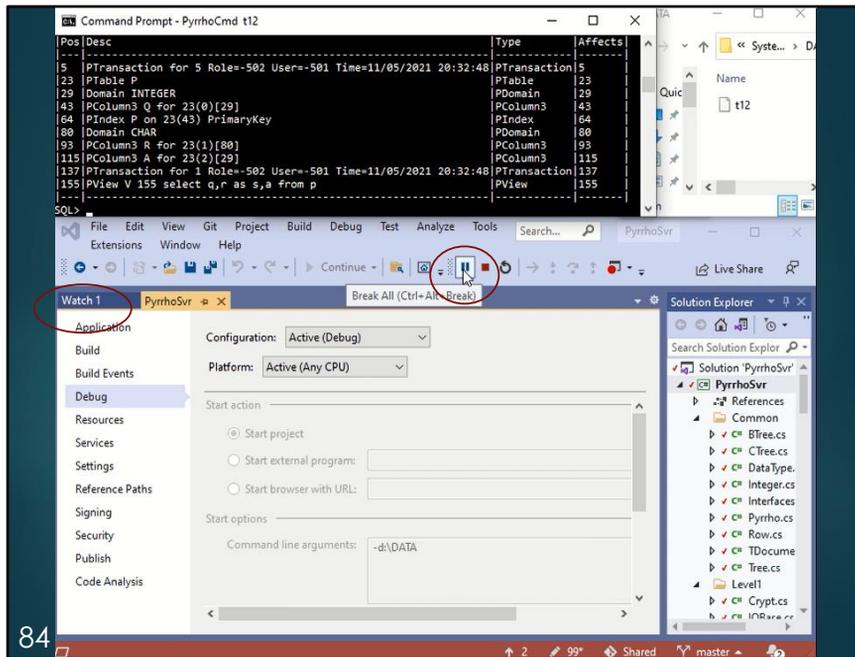


In the command window, enter commands to create a table P with three columns, and a View which uses this table, renaming a column. We also want to look at the transaction log.

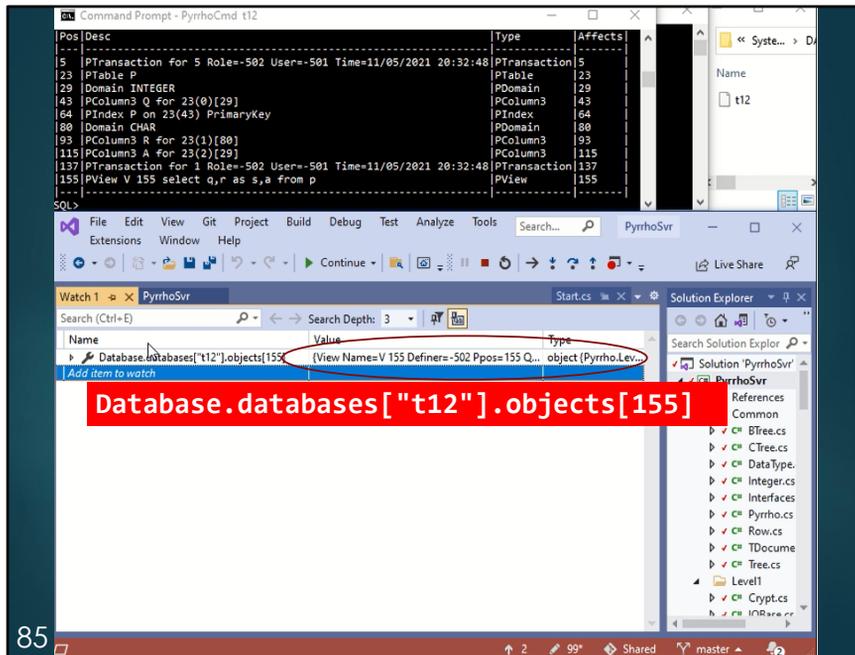


In the log file, we see that the View definition in the database file is just recorded as the source of the select definition of the view.

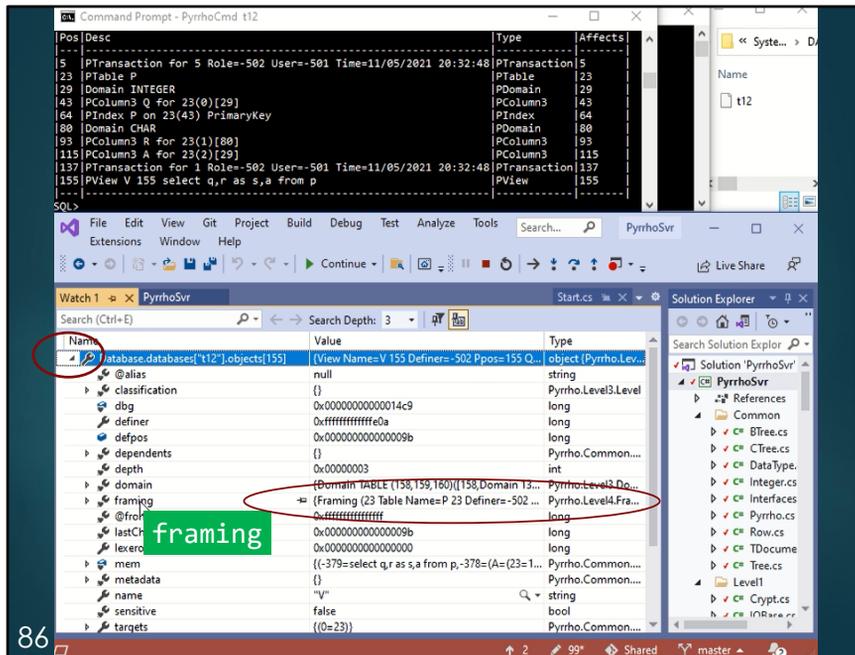
When the server processes this, on load or on definition, it creates some compiled components, so that this statement doesn't have to be parsed every time it is used.



Let's pause the server so we can look at the precompiled objects. Notice I have docked the Watch window as a tabbed document.

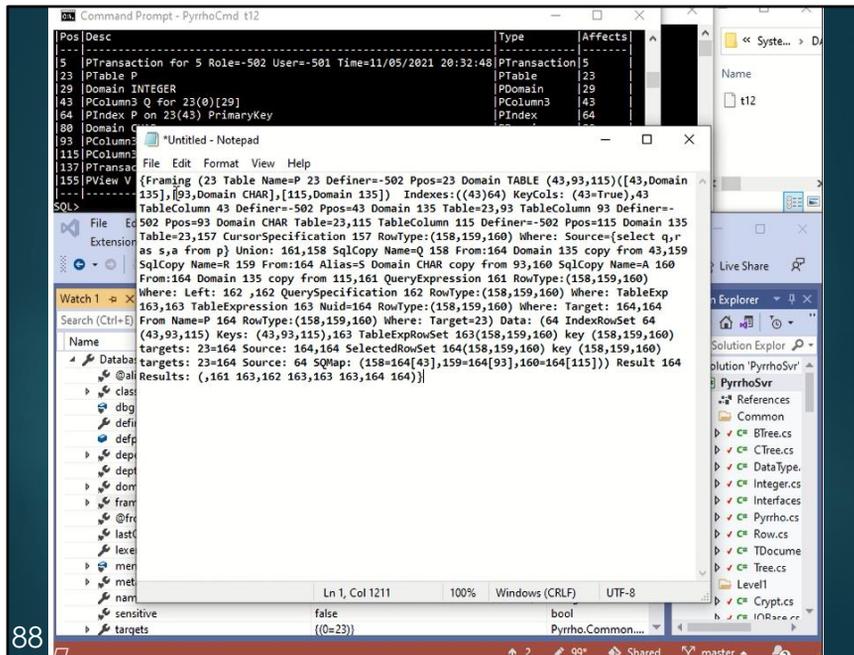


In the Watch window, examine `Database.databases["t12"].objects[155]` the View position. We see that there is a View defined in the Value.

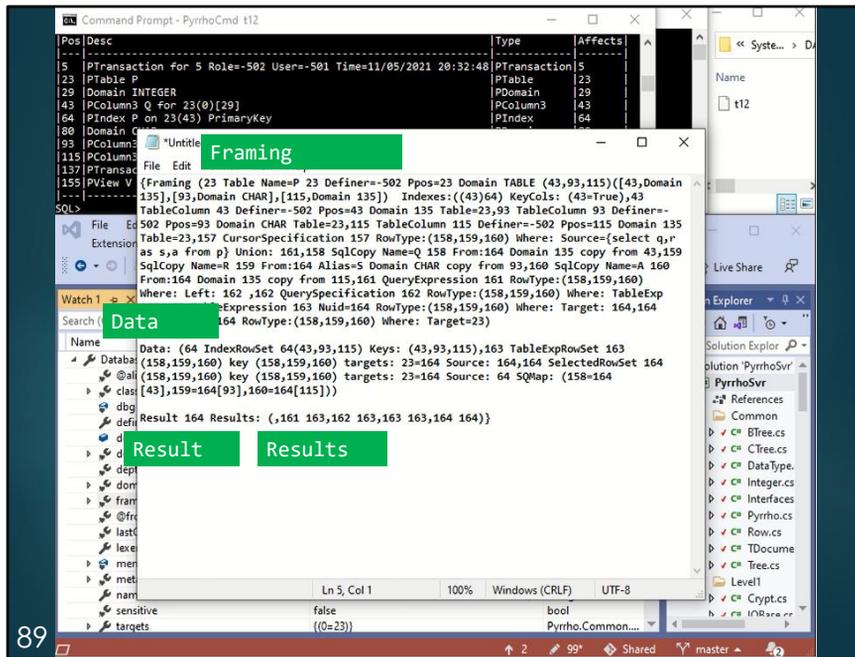


Let's expand that (it takes the server a moment to do it), and we see that there is a Framing field, and this contains the compiled components of the definition.

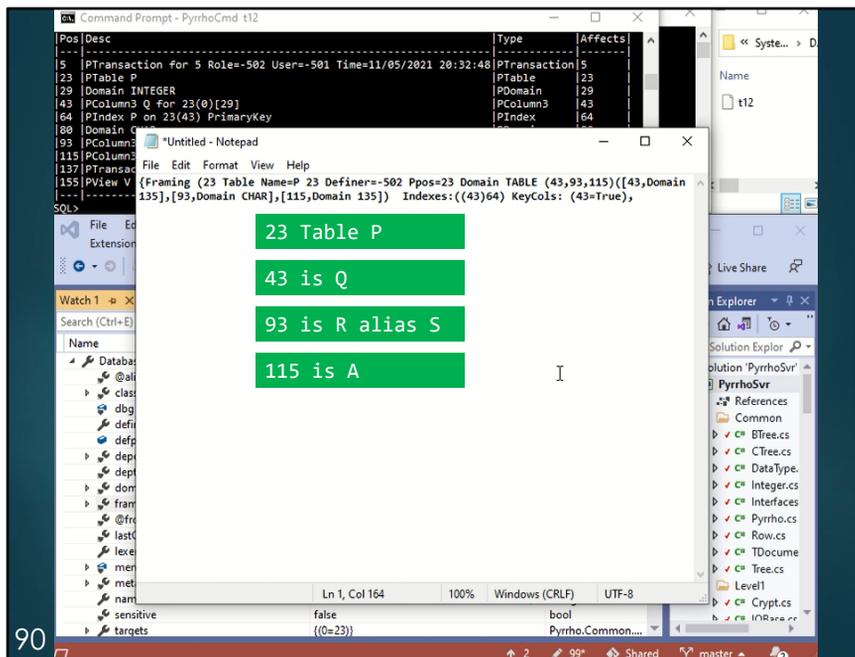




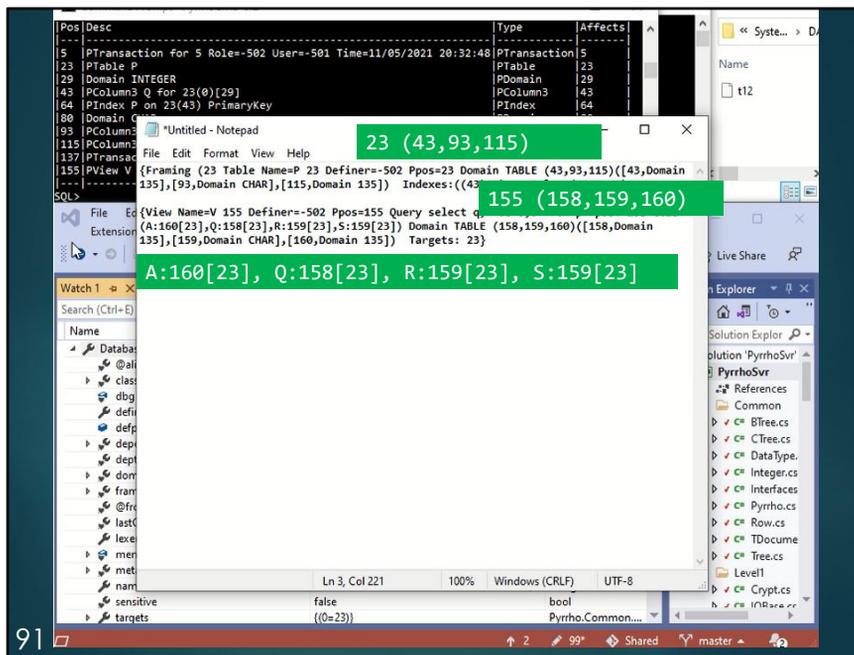
Open a Notepad window, and click Paste. We see there is a fair bit in the framing field. We don't need to look at all this detail. All we want to do here is to see that the Framing is in four parts. We add a little white space in the next slide



We can see there is a section of objects, a Data section for the rowsets, and a list of results.

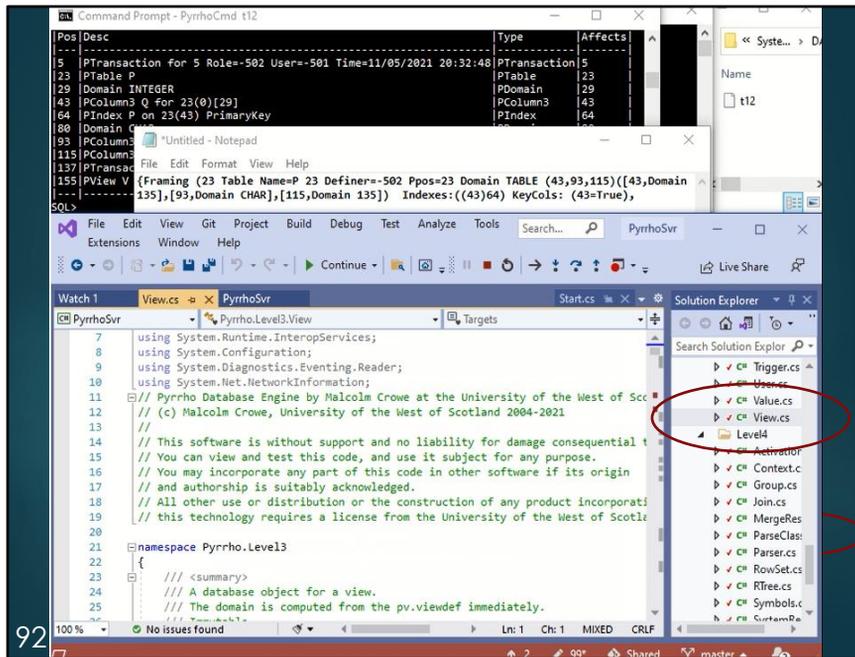


Just now, we will delete everything except the first two lines, which are the details for table P.

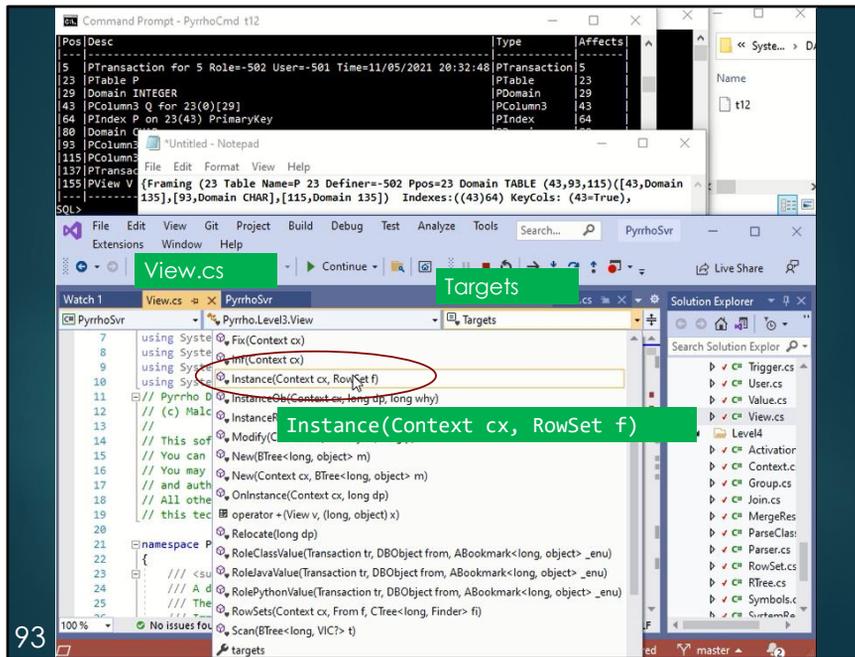


Let's also paste the View definition from the Watch window in here, so that we can see what's going on. We notice straightaway that the Domain of the View and the domain of the table don't match.

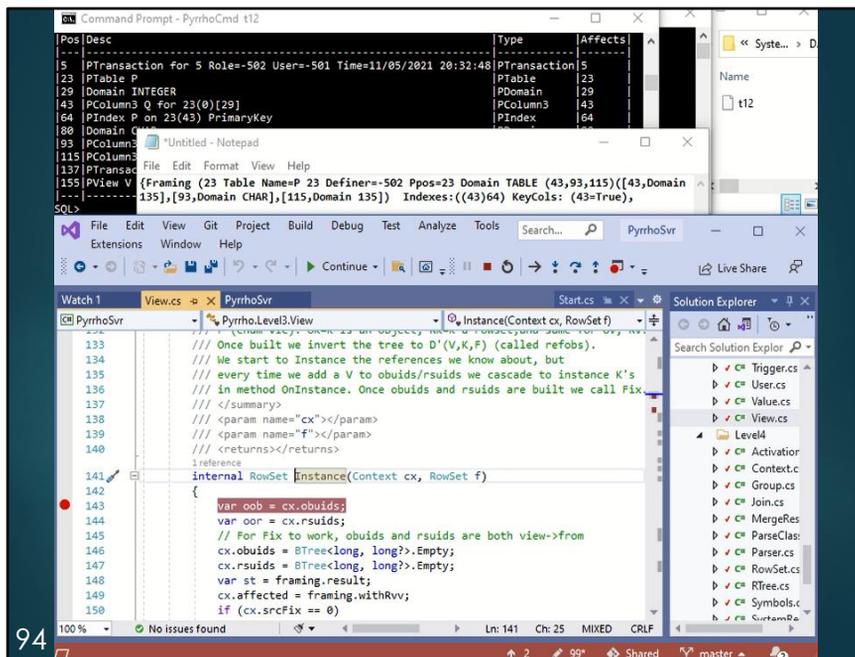
There is a relationship, as the types of the columns match, but the uids are all different. In the table, they are of the columns, but in the View these are some of the compiled objects.



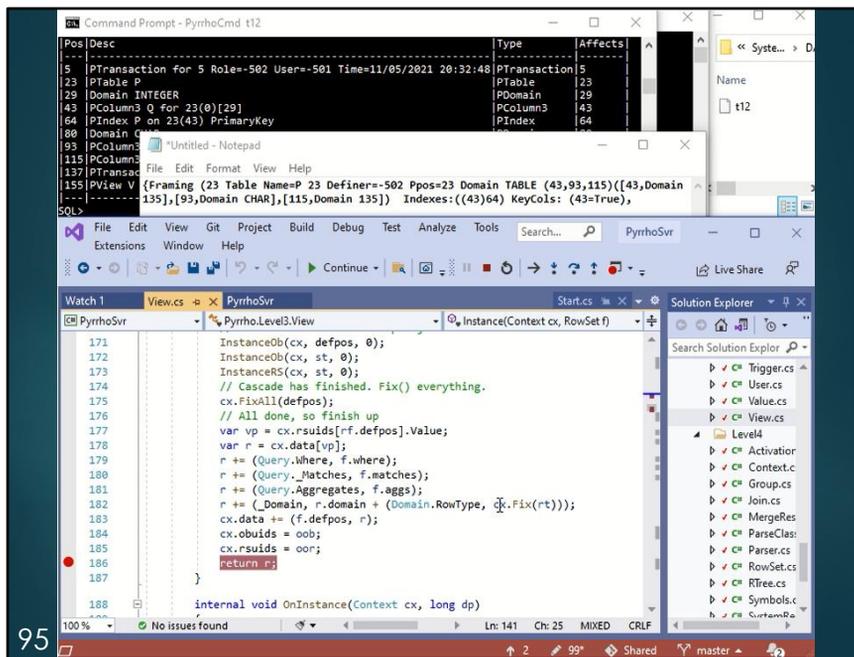
In Solution Explorer, scroll down to find View at the end of Level 3, and double-click.



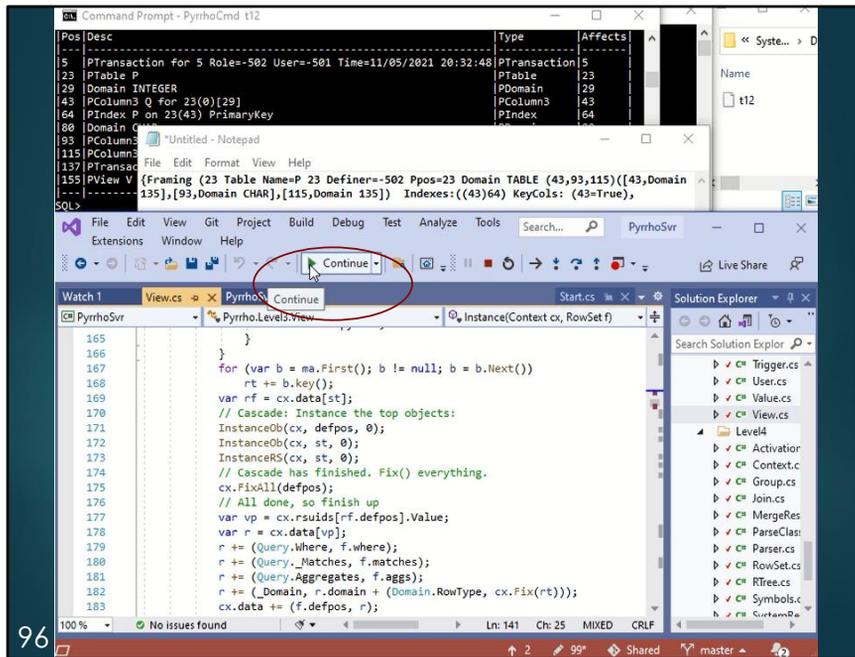
Find the Instance() method, and click on it.



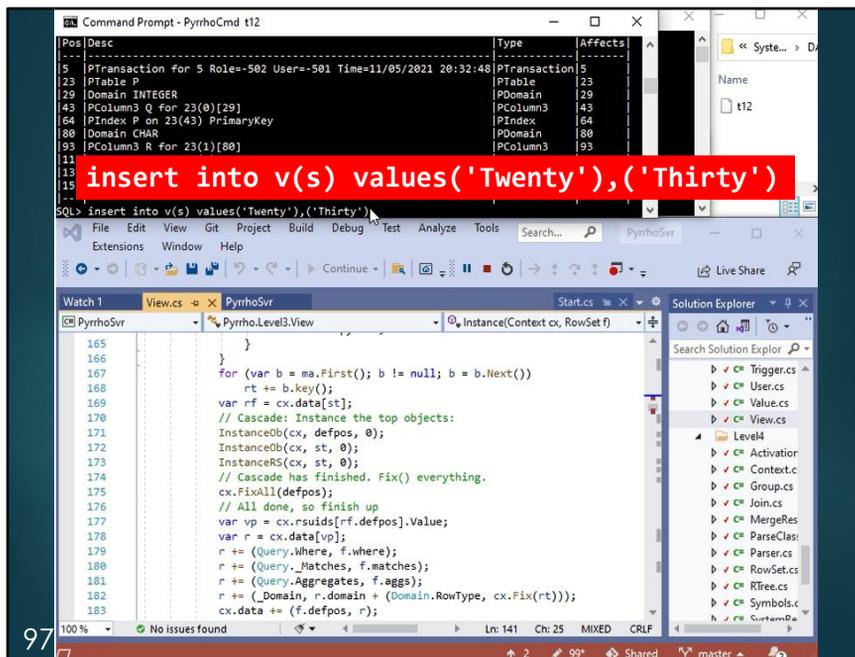
In the Instance() method, set a break point at line 143,



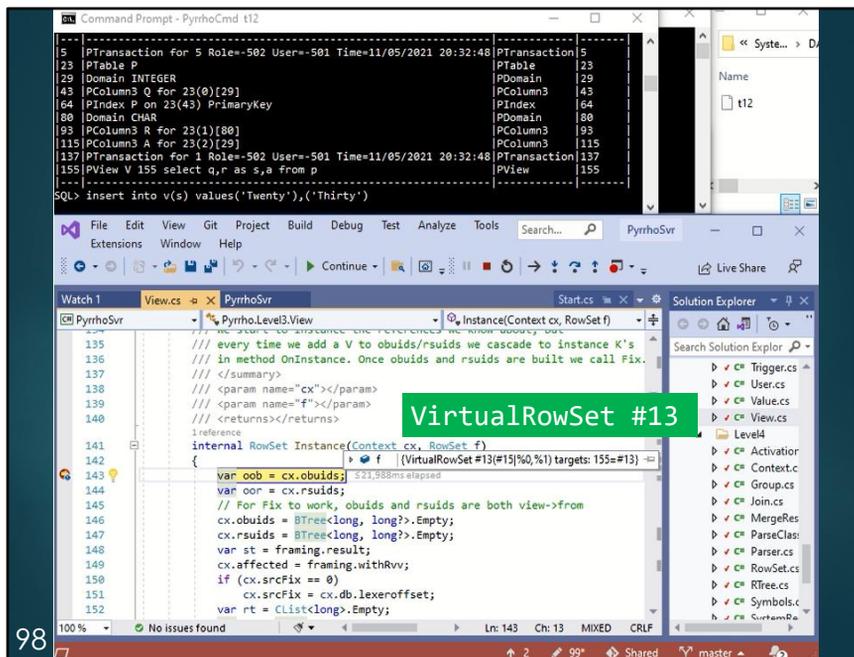
and also a breakpoint at line 186, at the end of the Instance method. The View.Instance() method responds to a reference to the View.



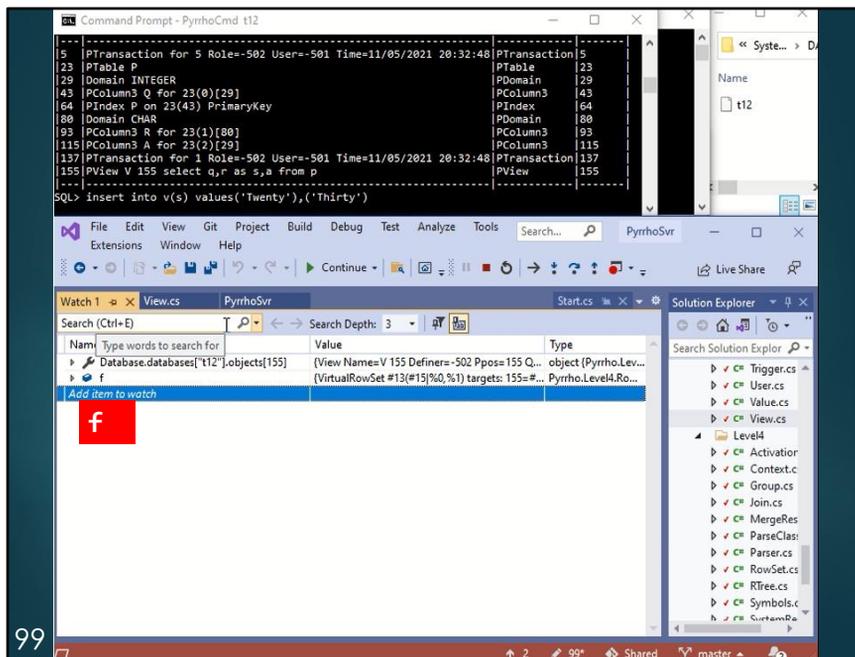
Click Continue, to allow the server to continue execution.



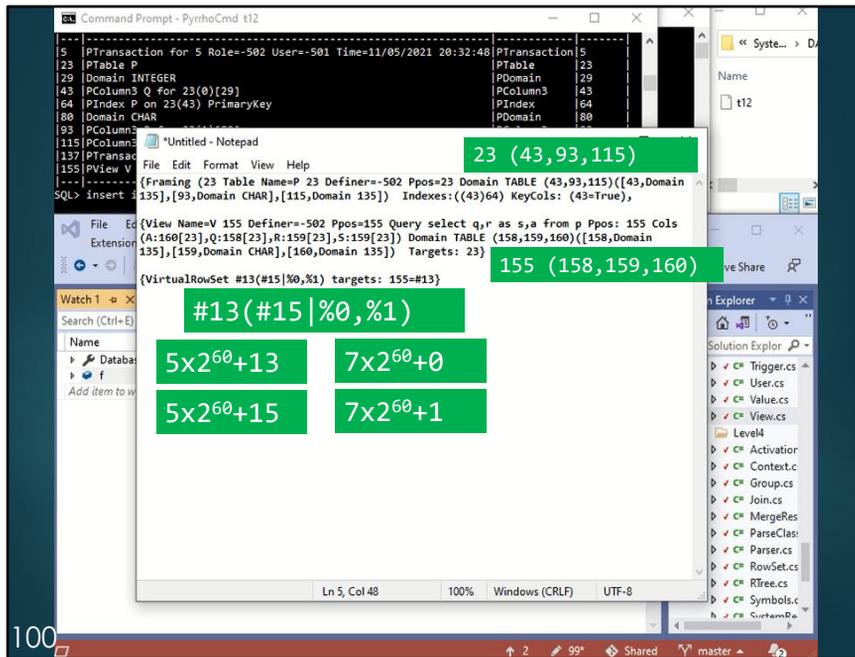
We are going to use the View definition to insert rows into the table P. This is obviously not what views are normally used for, but in this demonstration, we show that we have updatable views. Views can be used, provided the permissions are set correctly, to make modifications to the underlying tables.



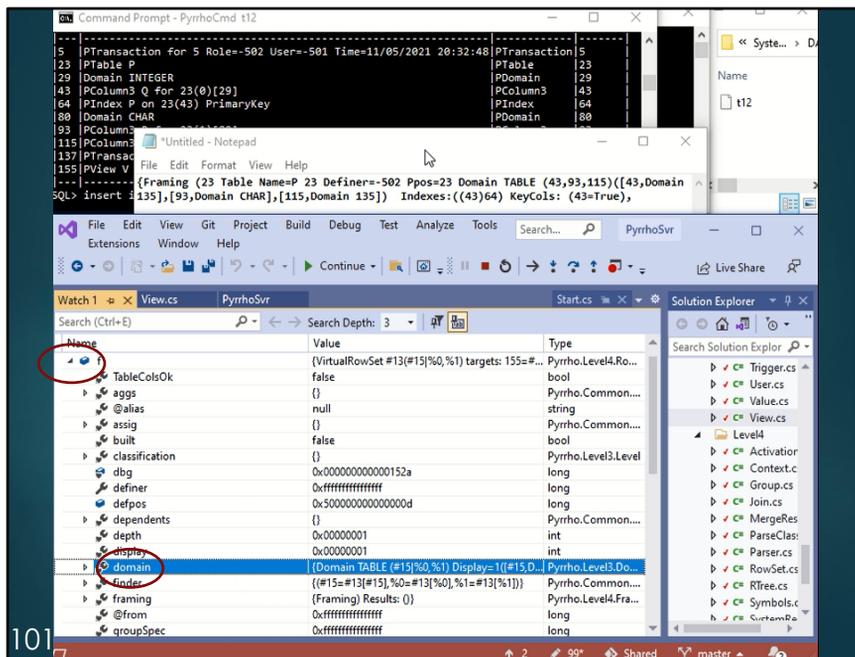
When we click Enter for this insert statement, Visual Studio stops at the break point. The Instance() method has a RowSet parameter RowSet f (its uid identifies the position of V in the command): it is a VirtualRowSet.



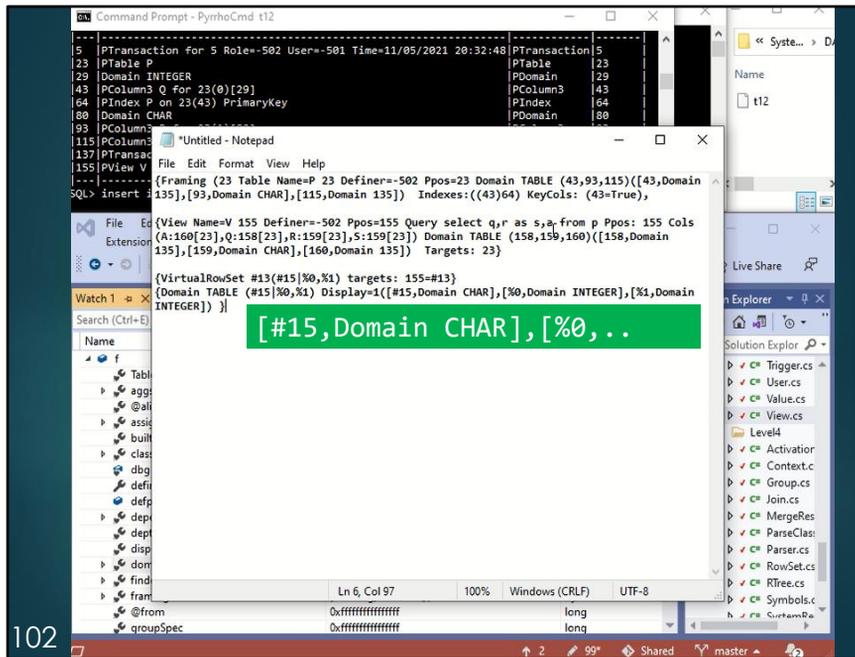
Use the Watch window to examine f. Right-click and copy its value into our Notepad.



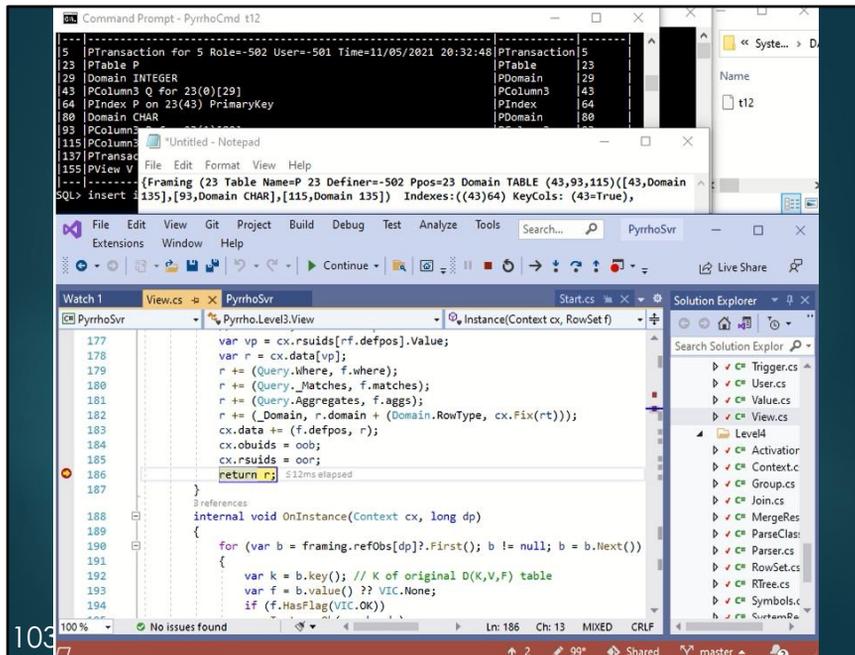
This domain seems different again, and the notation needs explanation. As mentioned, #13 is V's position in the command, and #15 S's position. These notations stand for long integers above  $2^{60}$   $0x500000000000000d$  and  $0x500000000000000f$  respectively, and %0 and %1 are  $0x7000000000000000$  and  $0x7000000000000001$  respectively. (We will soon see !0 which is  $0x4000000000000000$ .) These are long integers unlikely to clash with file positions.



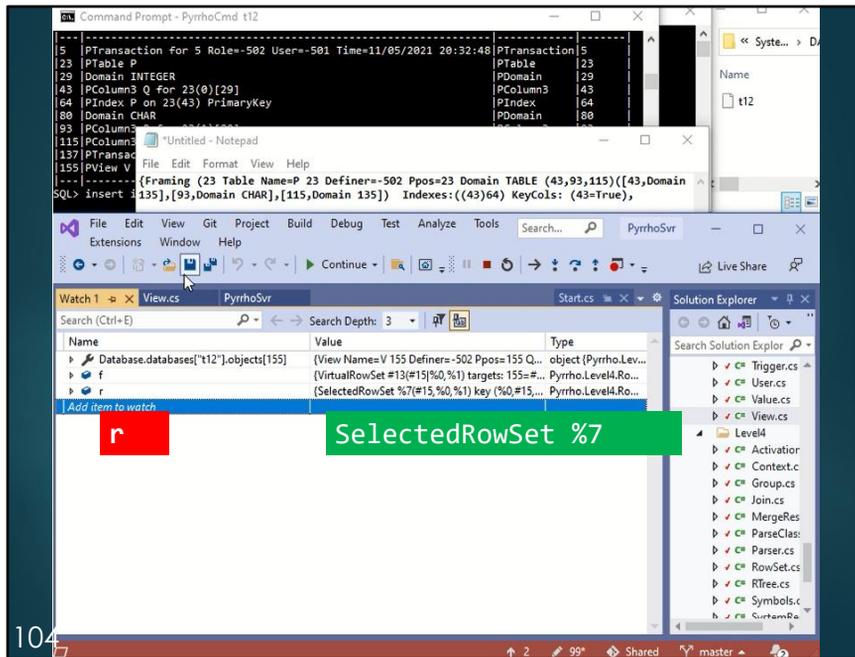
In the Watch window, expand f to get a closer look at the domain. Paste its value into the Notepad.



We notice that not only are the uids different, they are also in a different order. Continue to the next breakpoint.

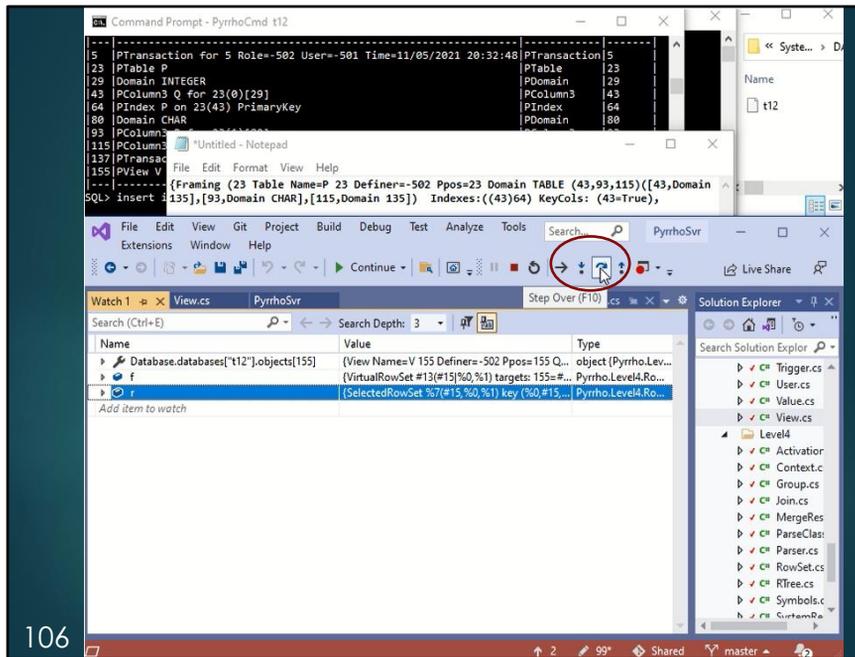


We are at the end of the Instance() method. Now examine the return value from the Instance() method in the Watch window.



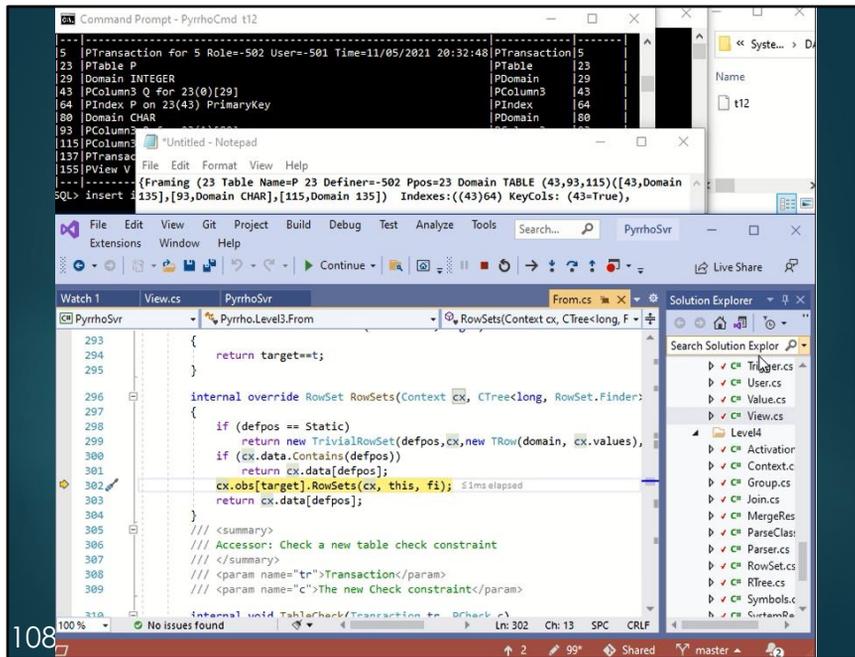
r is a SelectedRowSet. Paste its value into the Notepad.



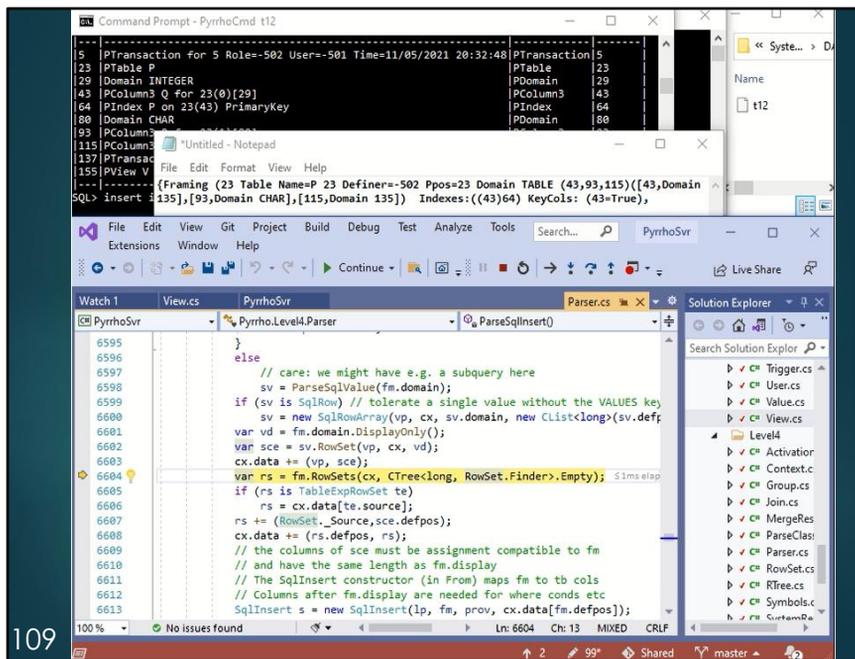


Back in Visual Studio, Step Over

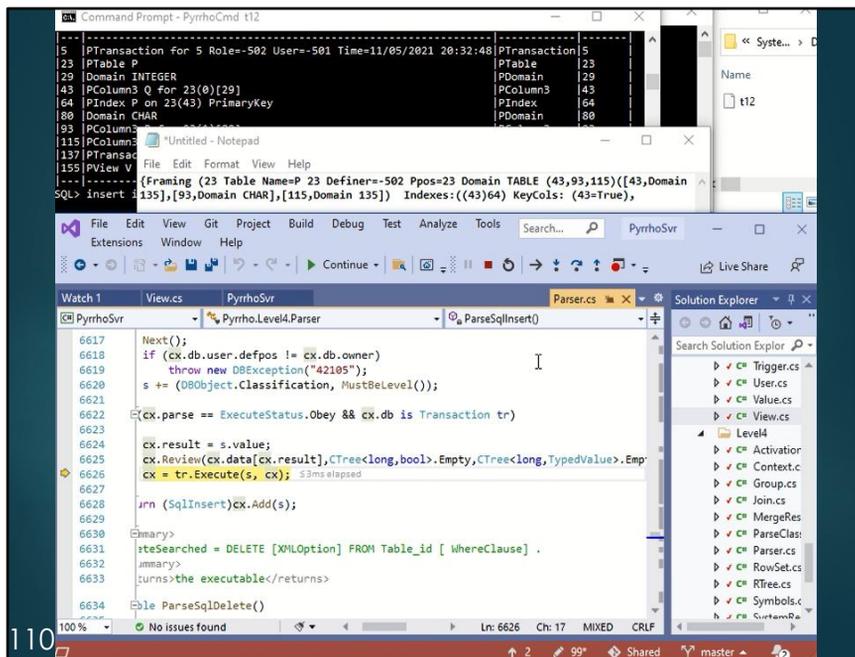




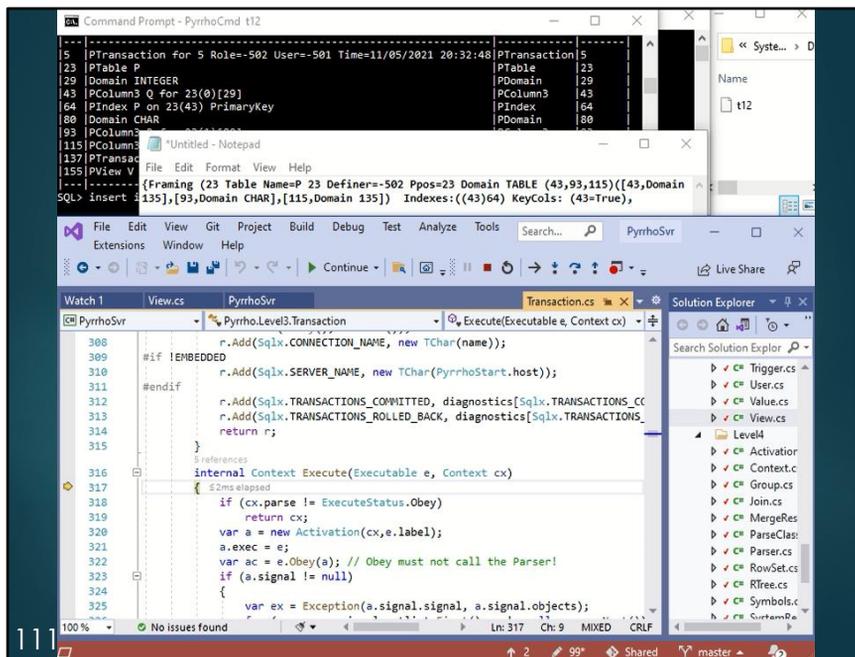
We are now back in `From.RowSets()`. Step Over



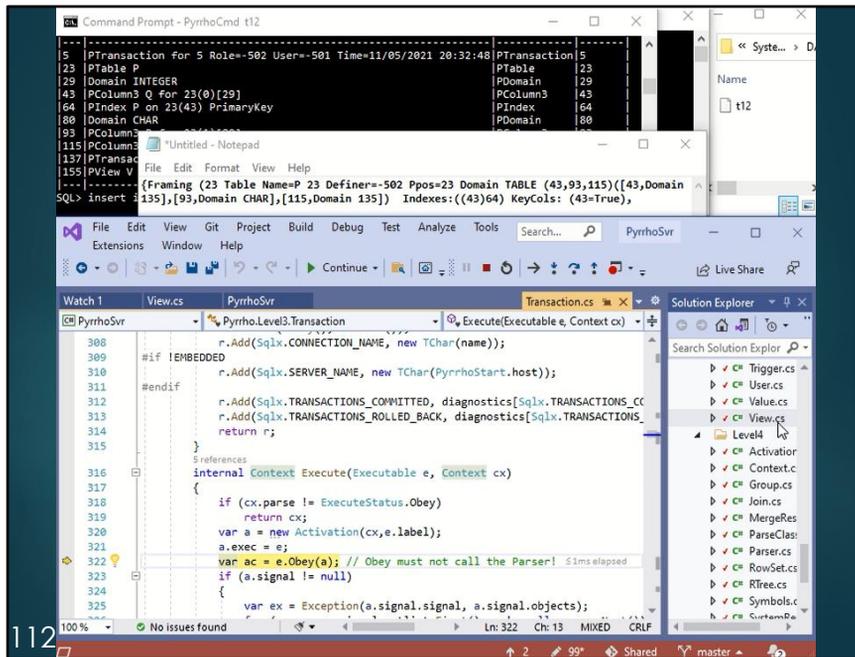
This takes us back to `Parser.ParseSqlInsert`. Step Over some more until line 6626.



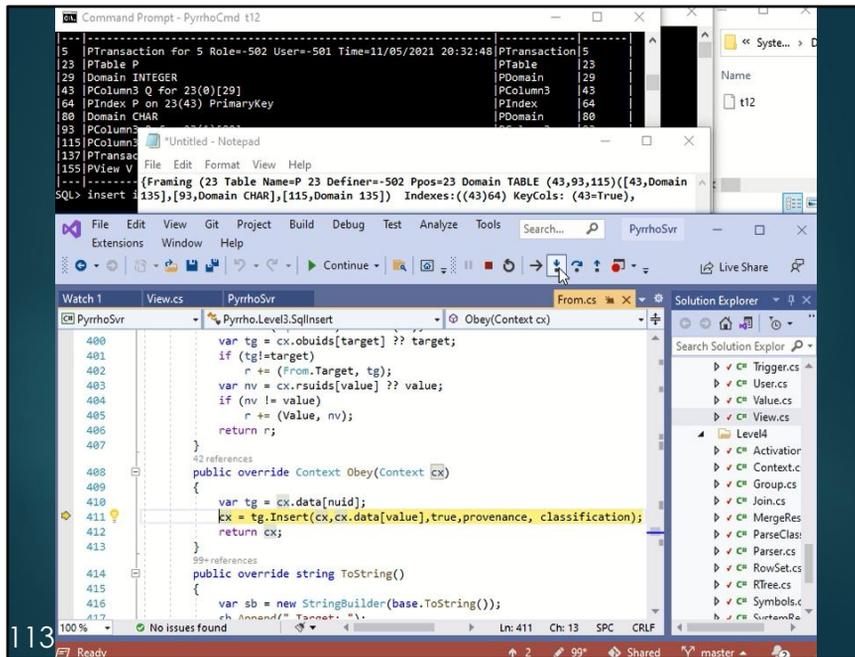
At line 6626, Step Into Transaction.Execute()



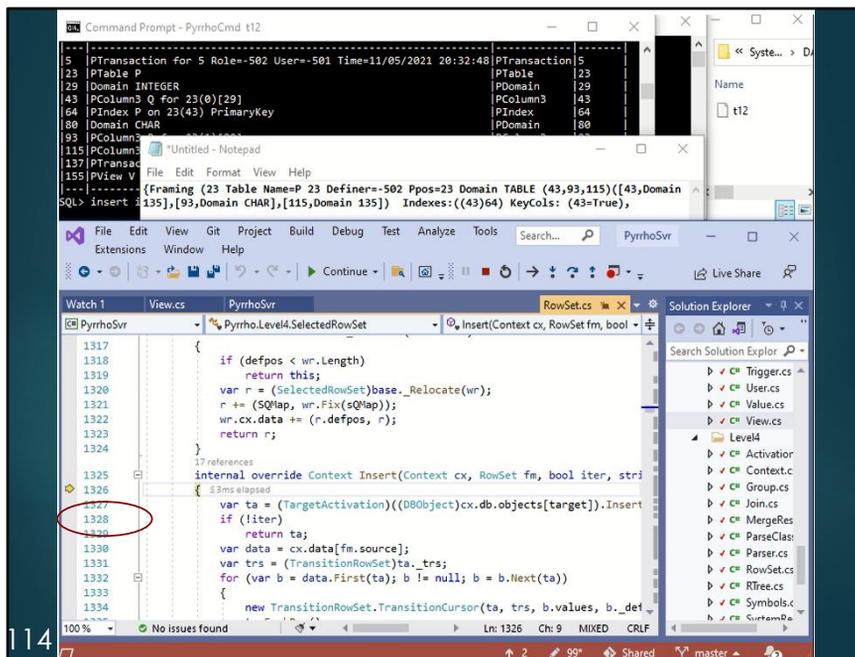
In Transaction.Execute(), Step Over until line 322.



At line 322, Step Into Obey()



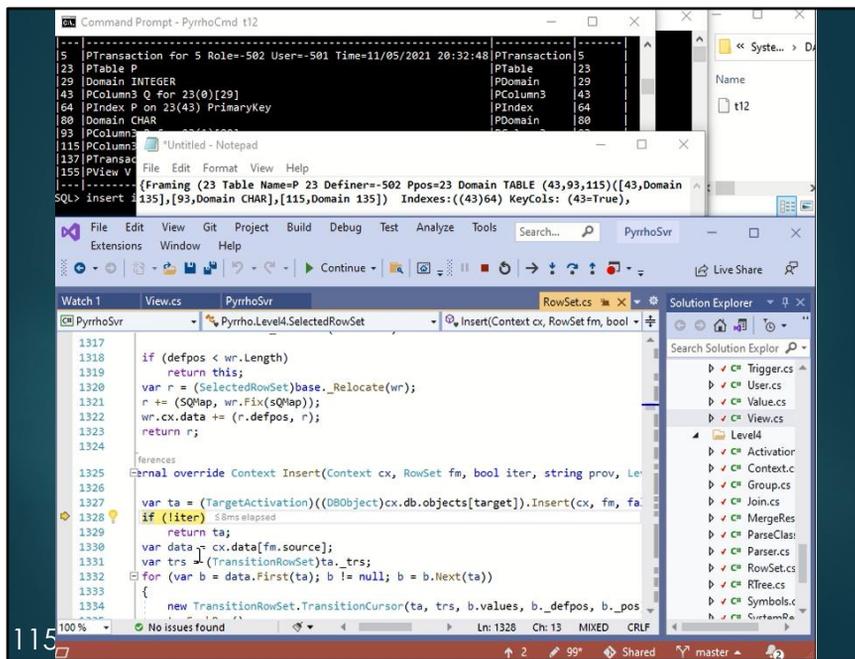
In `Sqlinsert.Obey`, step into `Insert()`



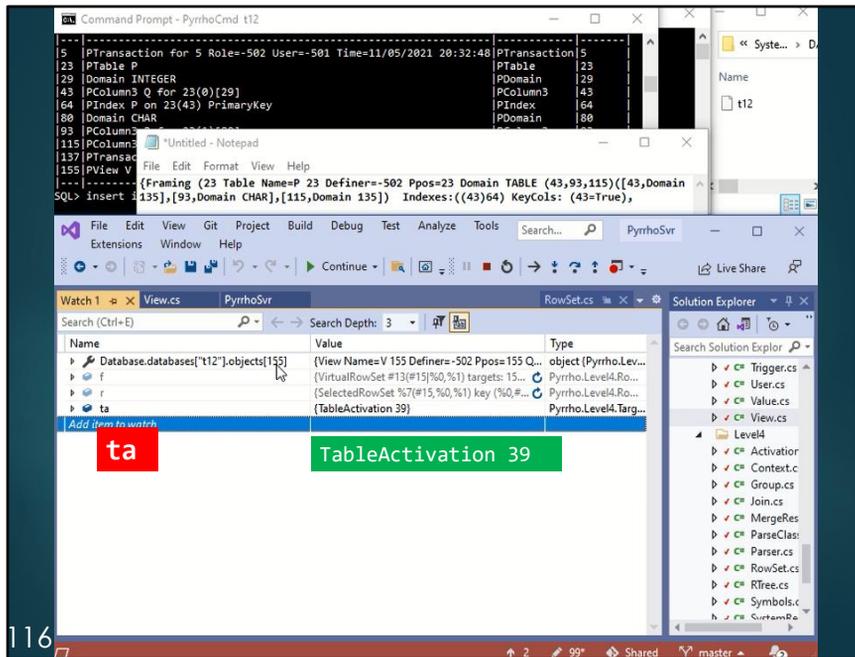
114

This is where the work is done.

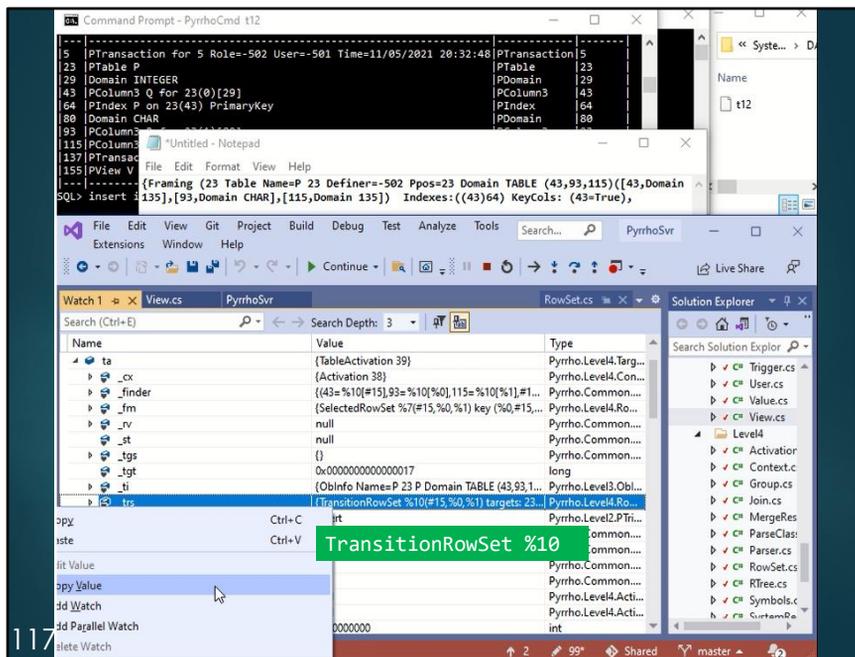
Inserting into a base table often involves Triggers. Triggers are managed by Activations. Step Over the creation of the TargetActivation, to line 1328.



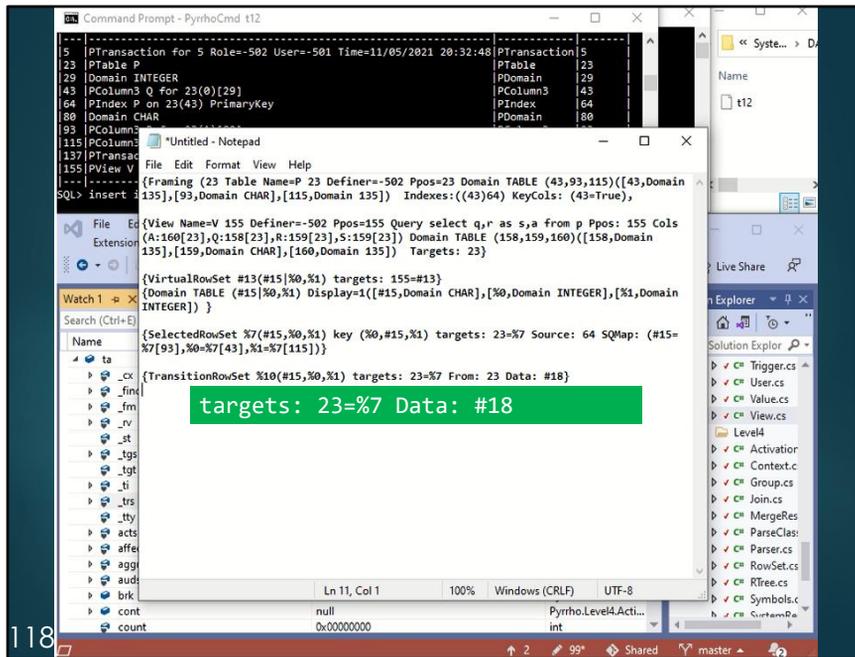
At line 1328, use the Watch window to examine ta



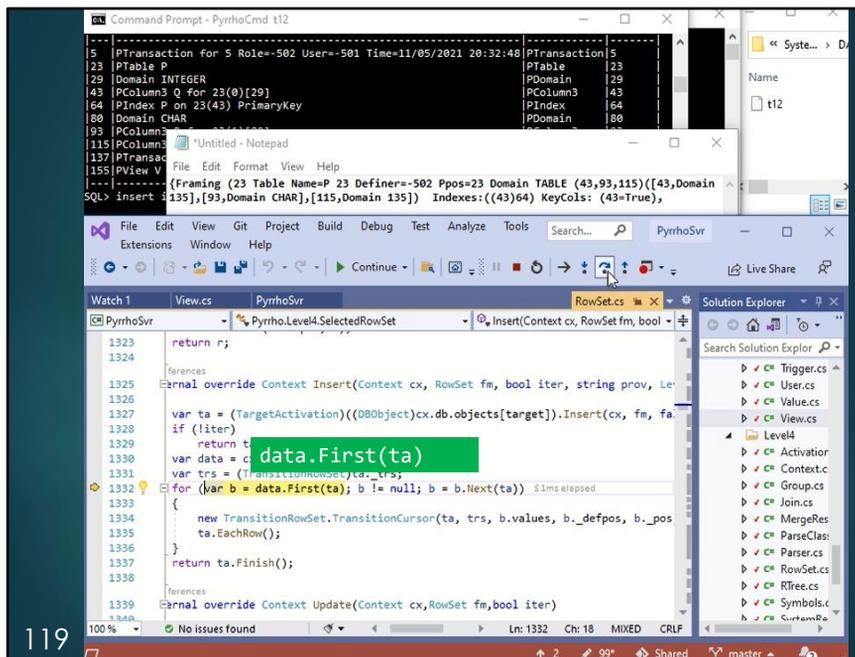
The source code said this would be a TargetActivation. It is a subclass, TableActivation. Expand it.



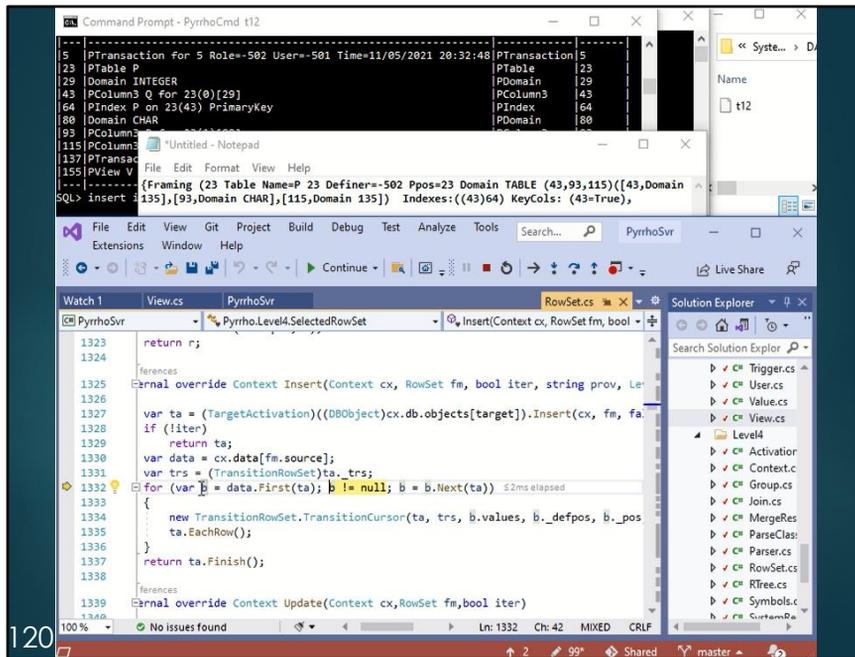
The TableActivation uses a thing called a TransitionRowSet, a concept defined in the SQL standard. Copy its value into our Notepad.



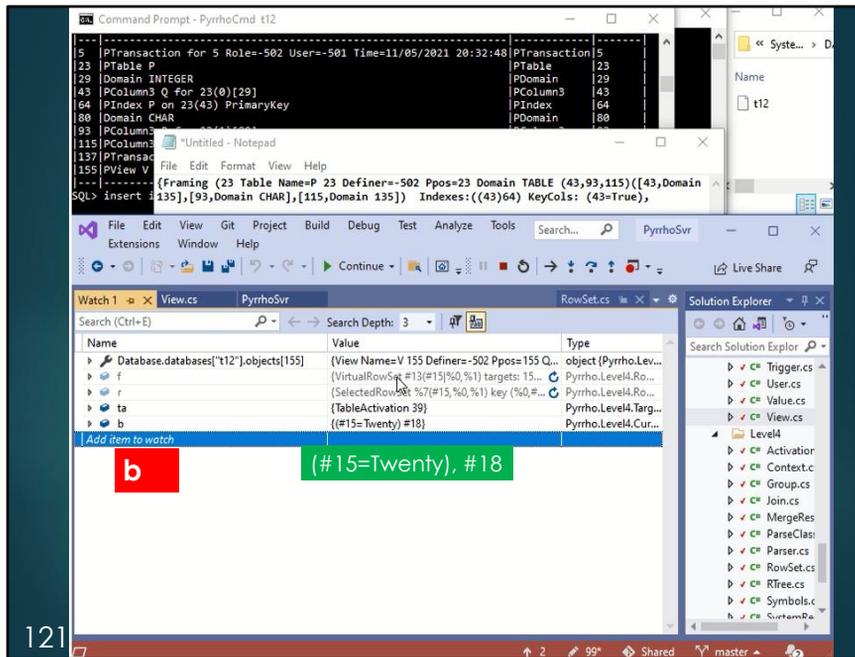
We see the TransitionRowSet knows it has Data, VALUES is at position 18 of the INSERT statement.



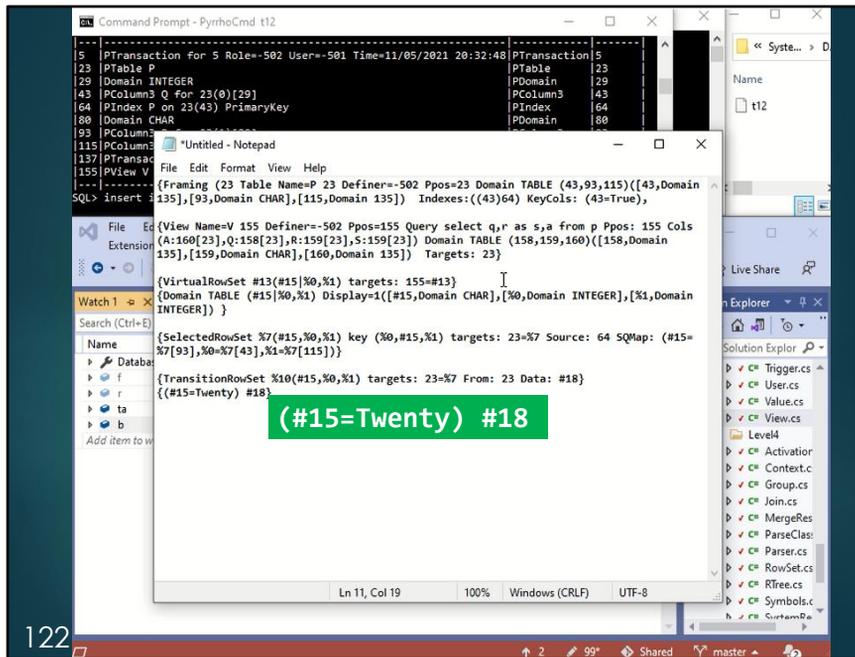
Back in the Insert method, start traversal of the data.  
Step Over the call to First().



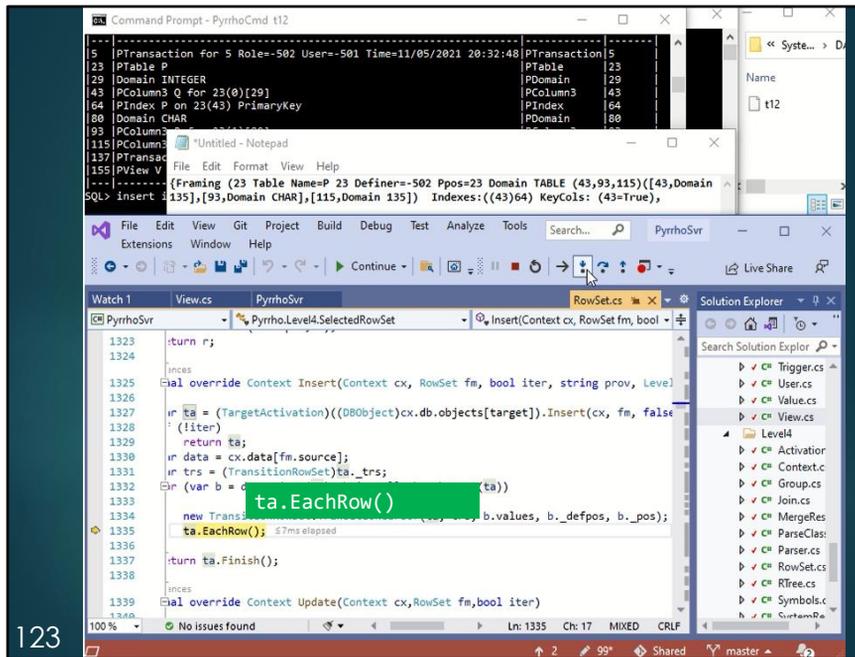
Now use the Watch window to examine Cursor b



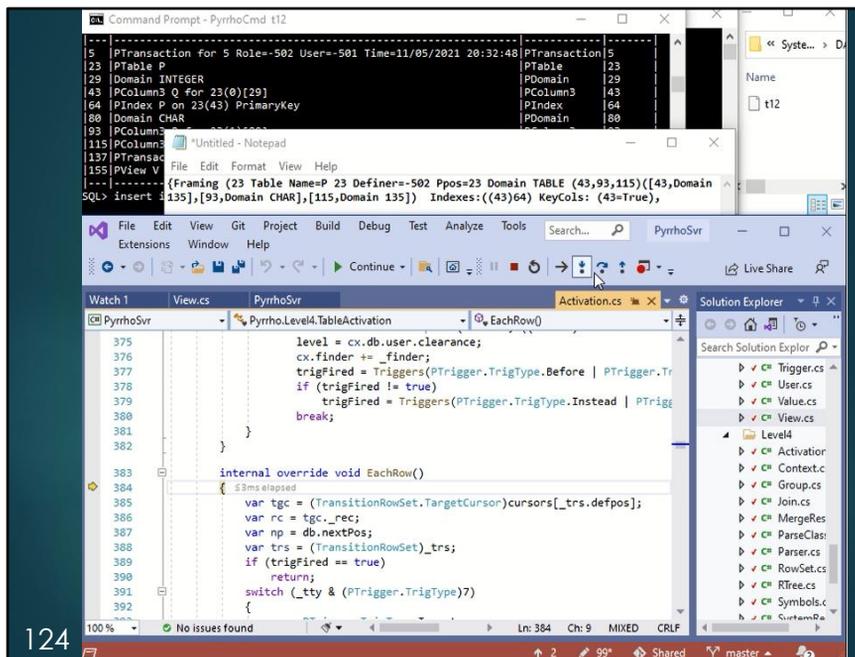
Paste the value of the cursor into the Notepad



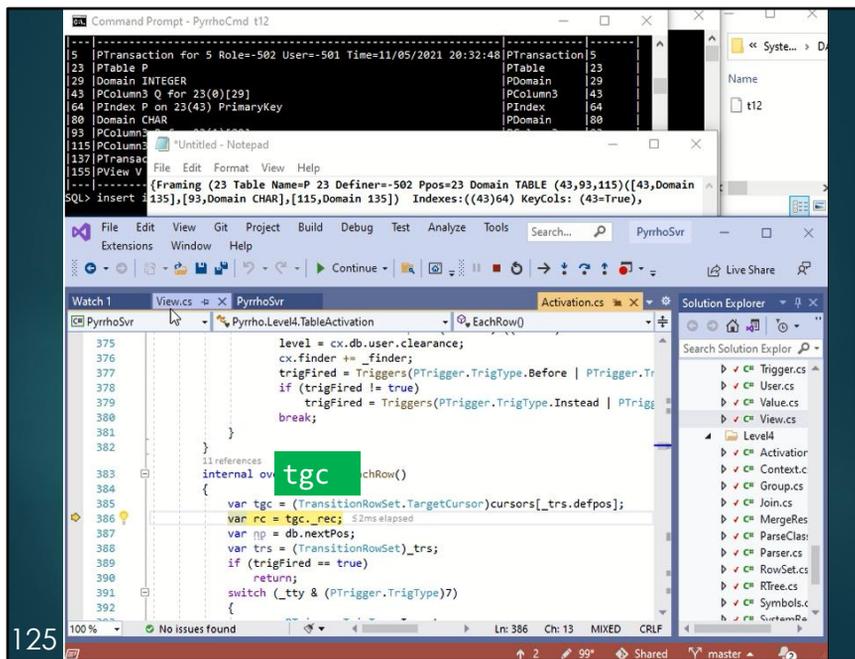
We recognise this value from the INSERT command. Step Over



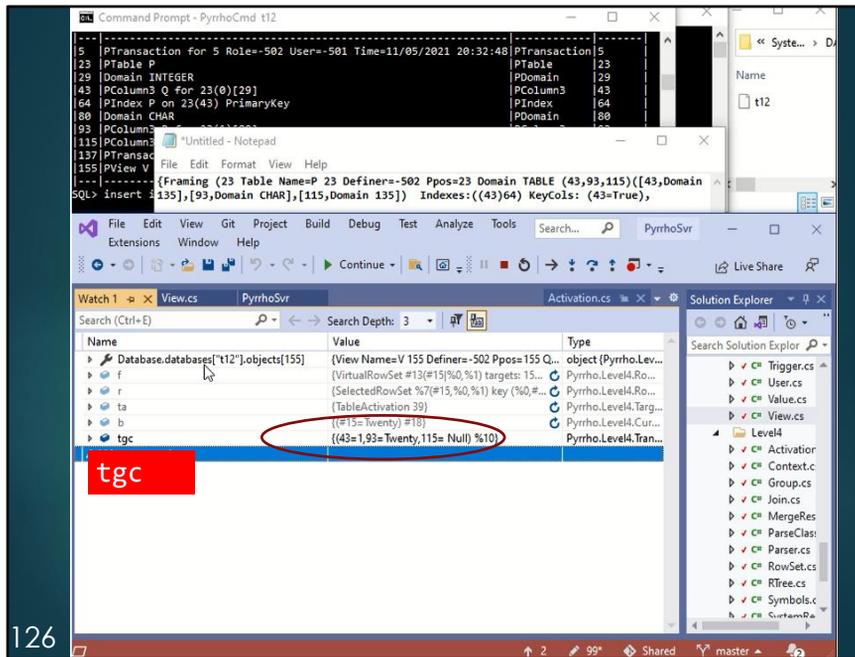
Now Step Into EachRow()



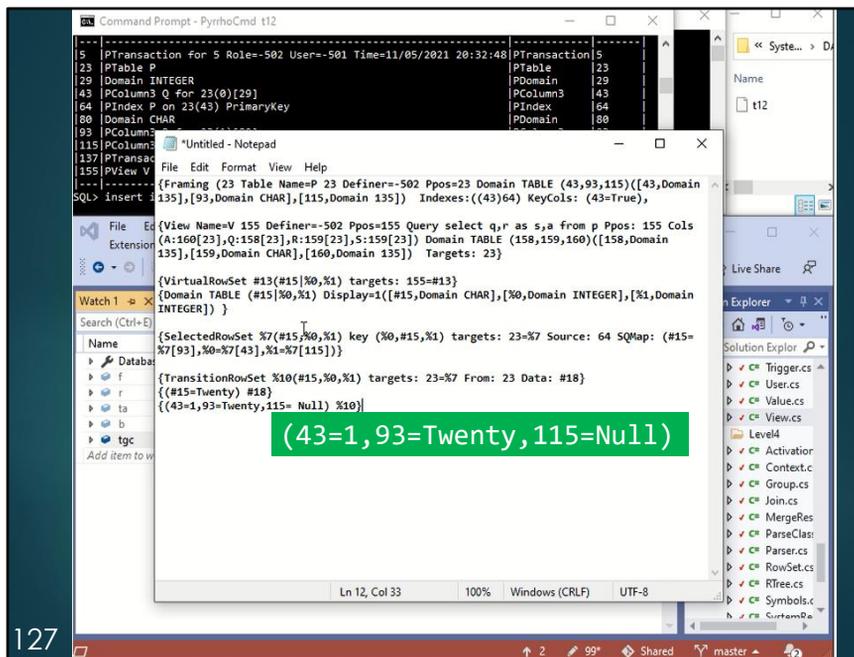
In `TableActivation.EachRow()`, Step Over twice



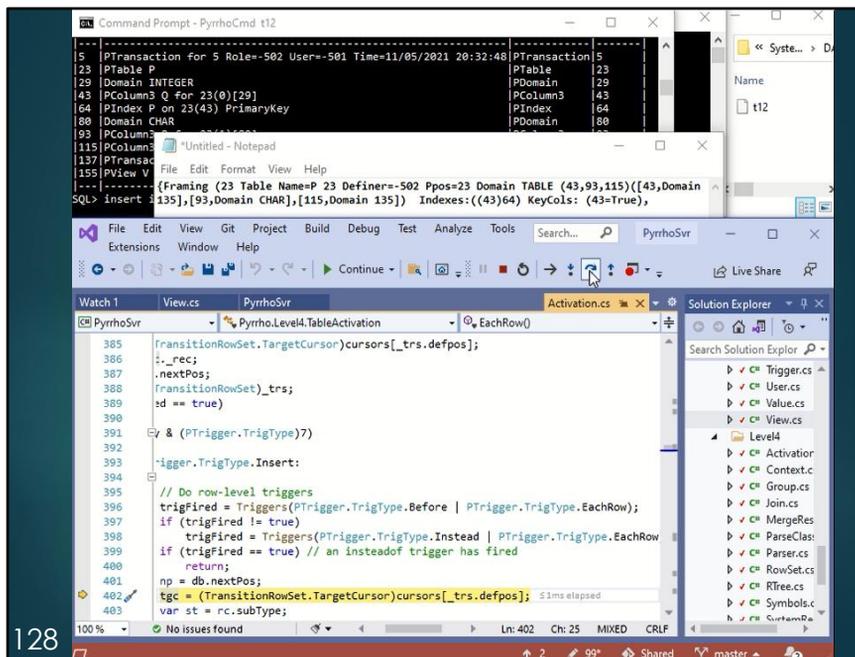
At line 386, use the Watch window to examine the TargetCursor tgc



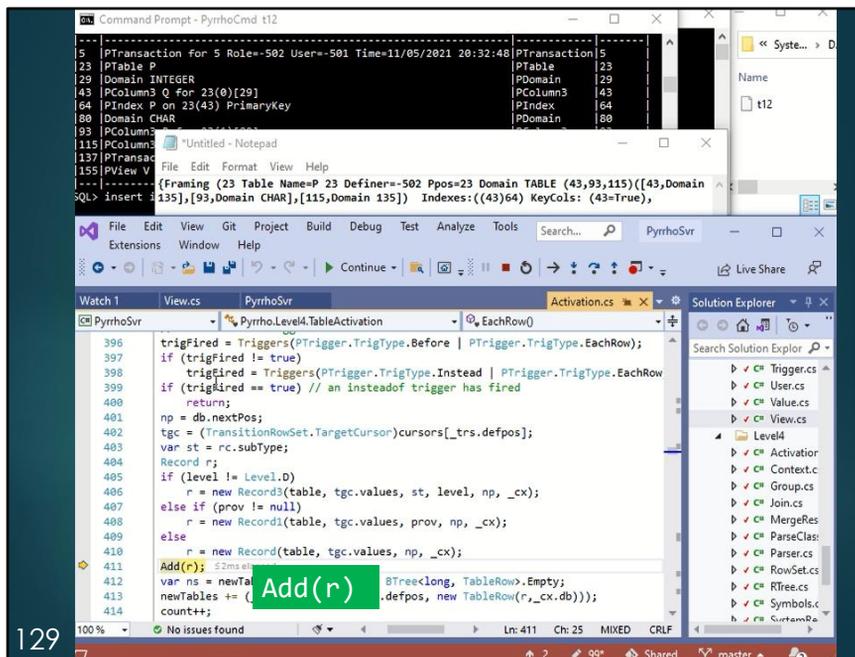
Paste the value of tgc into the Notepad



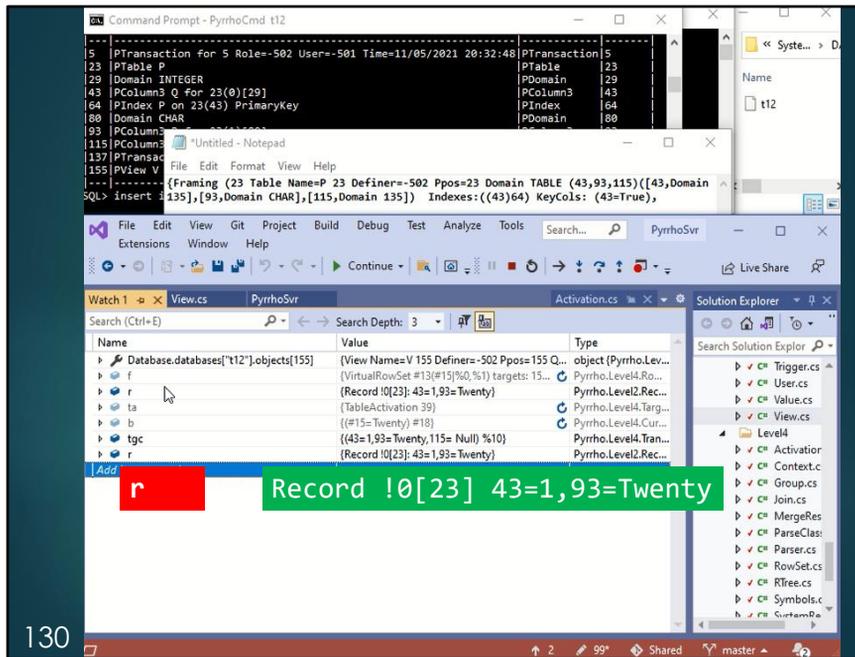
Here we see that Pyrrho's autokey feature has filled in a value for the key column 43.



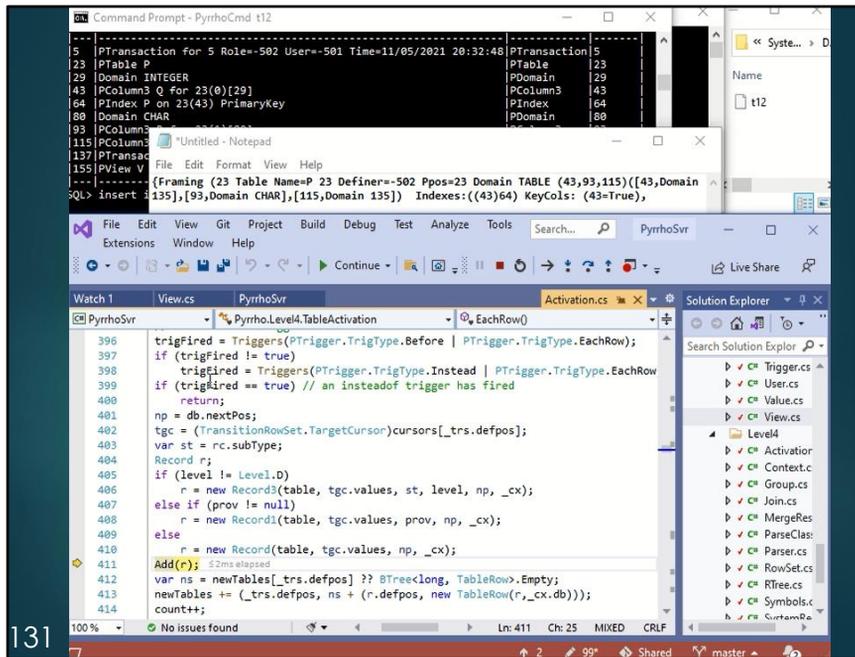
Step Over a bit more. We have no triggers in this example, so after refreshing the value of the target cursor, Pyrrho constructs a record. We will stop at line 411.



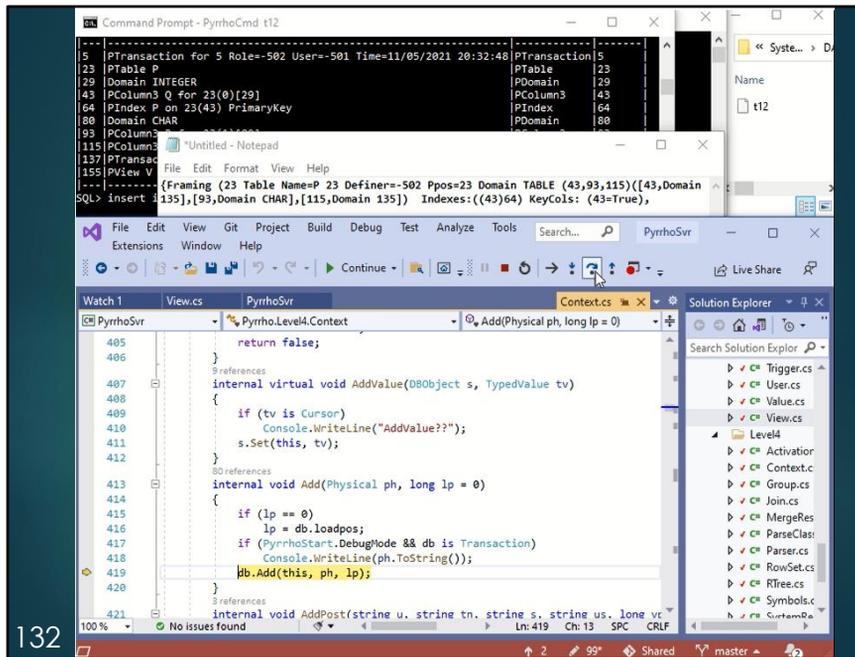
At line 411. A Record r has just been constructed.



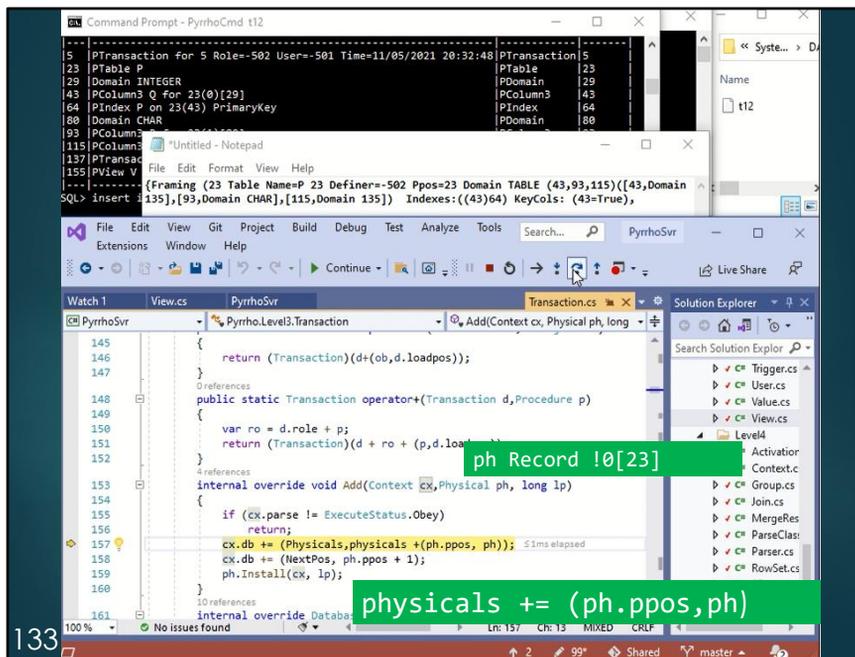
Use the Watch window to examine r. From now on it is all about installing this record in the Transaction.



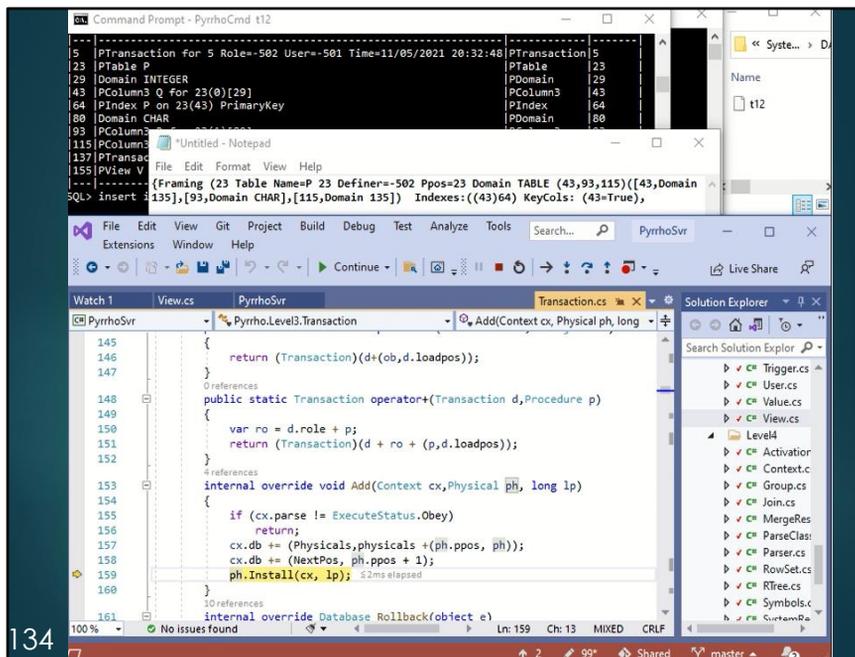
Back in Visual Studio, Step Into Add(r).



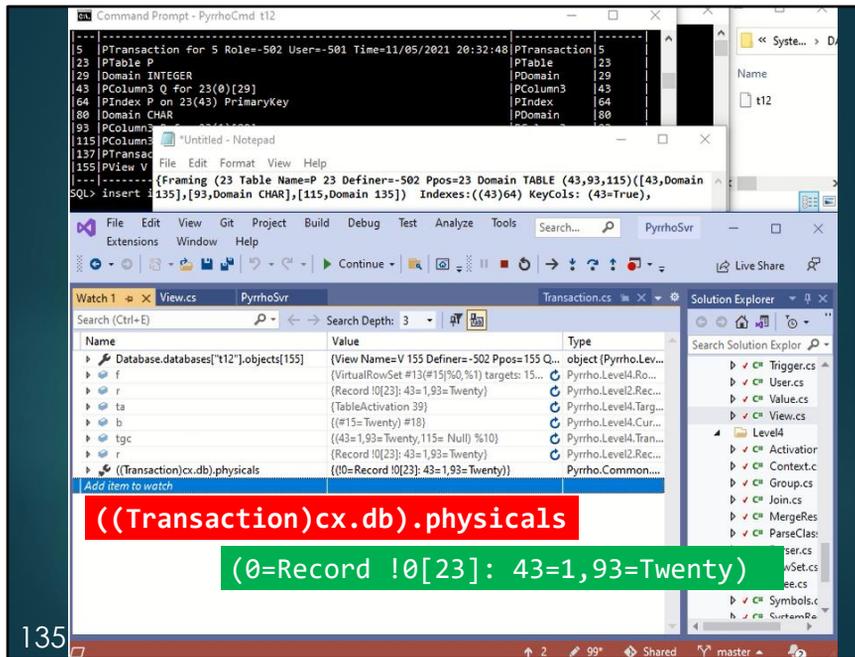
In `Context.Add()`, step over a few times, then step Into `db.Add()` to add our Record to the Transaction.



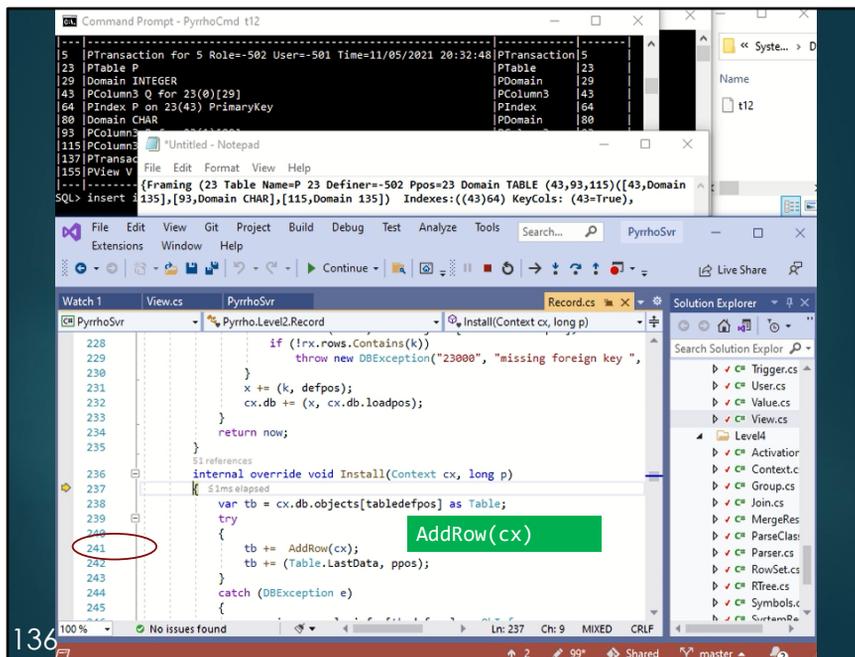
In Transaction.Add, we see how the Physical is added to the physicals list. Step Over line 158.



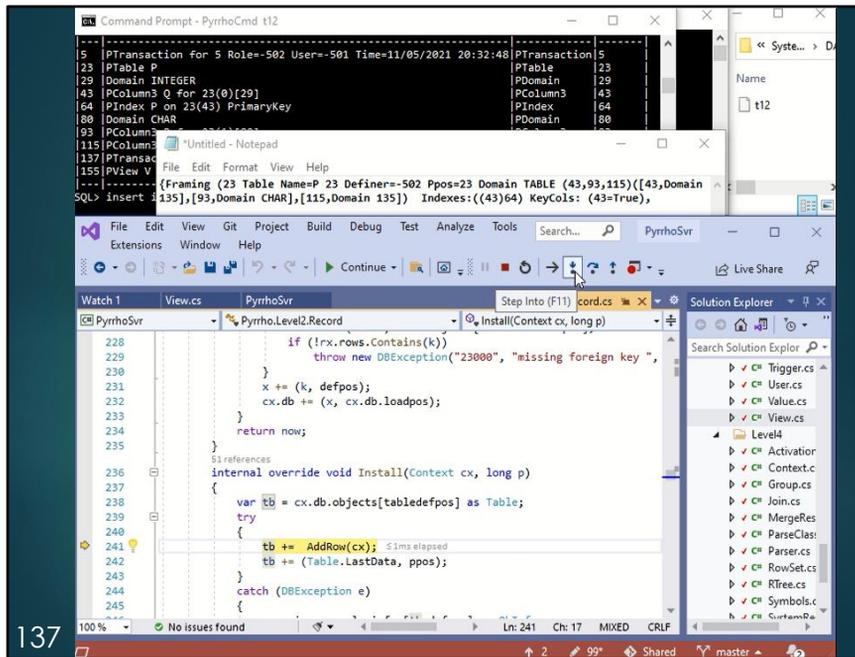
At line 159, we are about to call `ph.Install()`. But let's use the Watch window to examine physicals



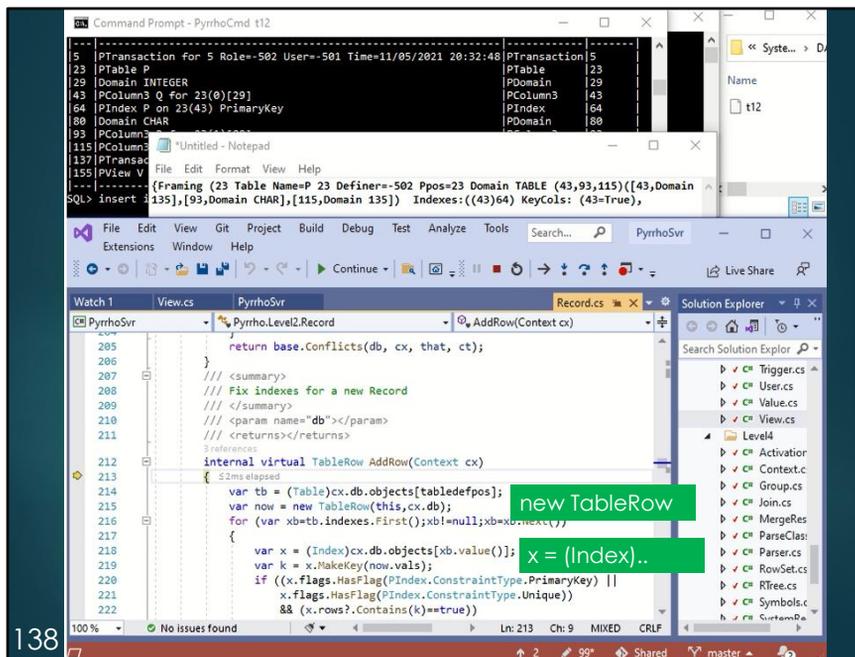
It is a list containing the new Record. Step into  
ph.Install()



In Record.Install(), Step over to line 241

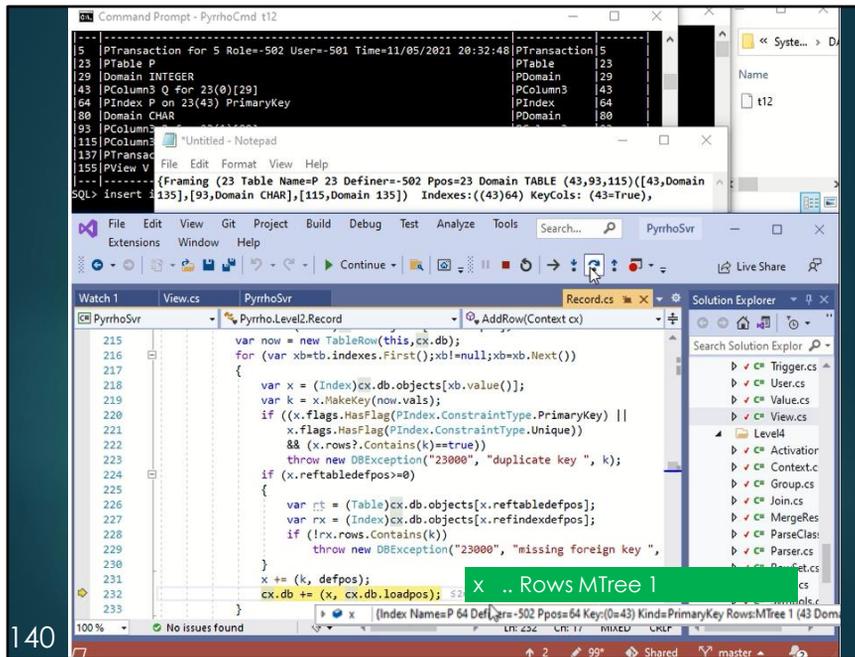


Step Into AddRow().

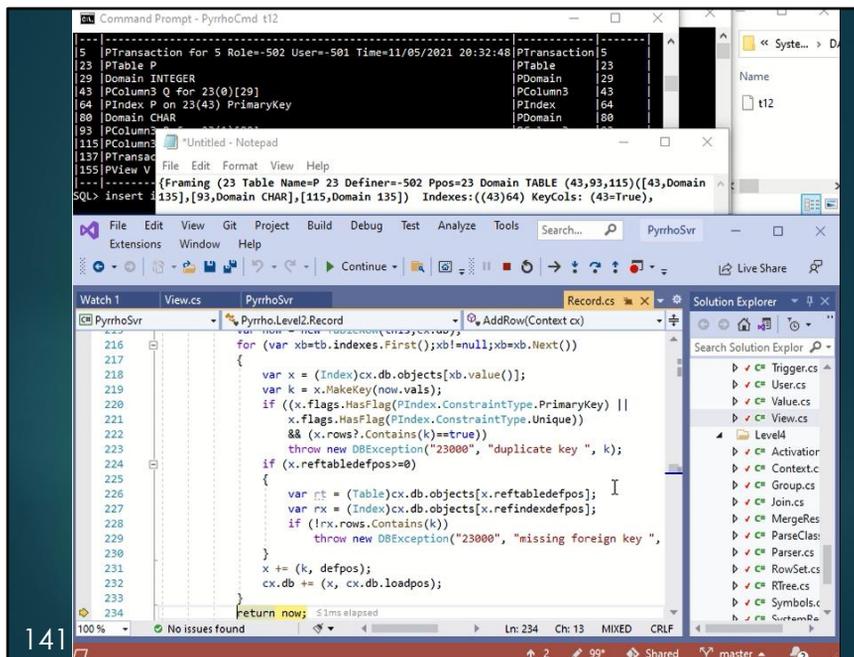


Record.AddRow is all about building a TableRow and adding it to any indexes that are around. Step over down to line 231.

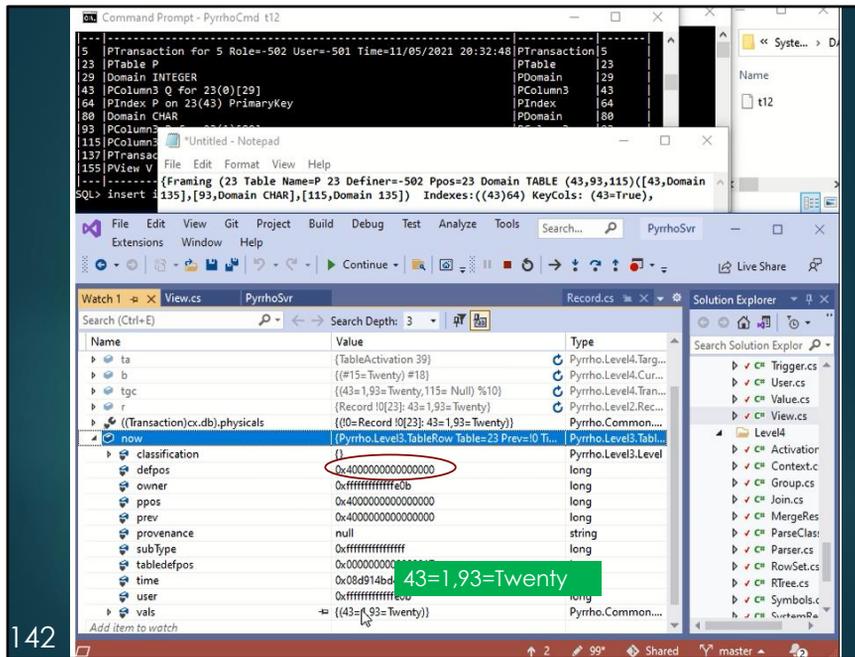




Hovering over x, we see x has a new value and now contains one key. Another addition operator installs the new Index in the transaction.

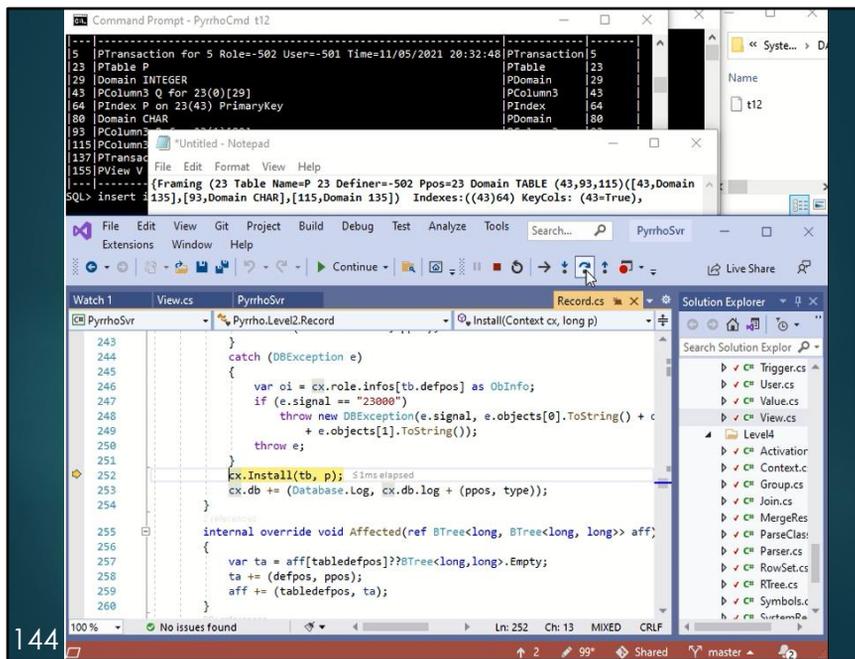


After a journey around the loop we reach the return statement at line 234 with the new TableRow.

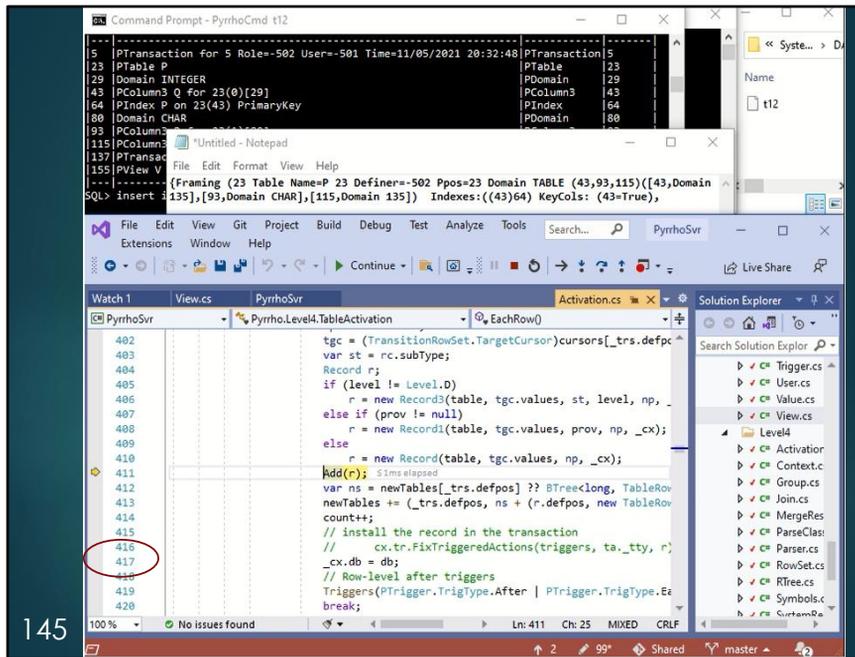


Let's examine now with the Watch window. It has our new values vals. The huge numbers called ppos etc are all !0, the position for the first uncommitted database object in the transaction.

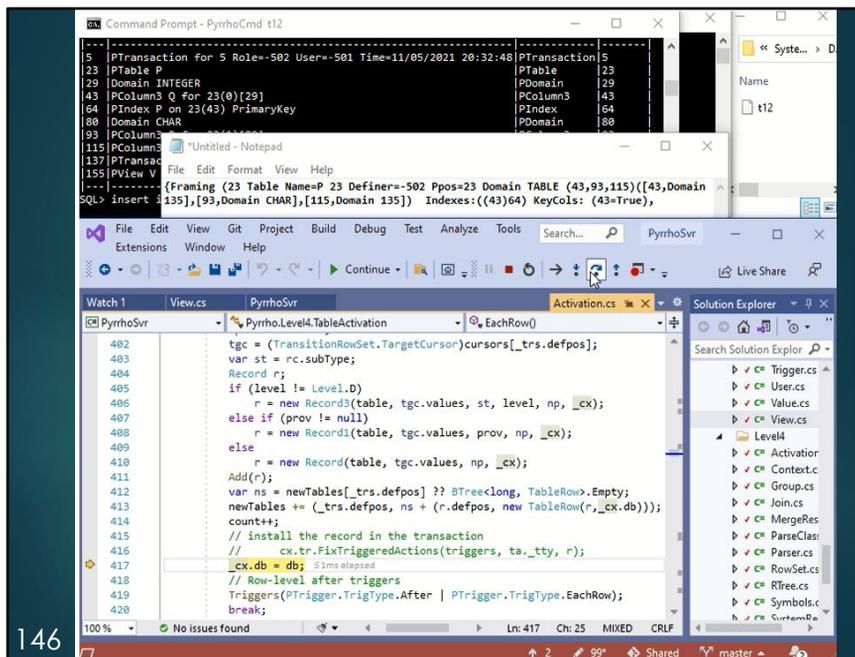




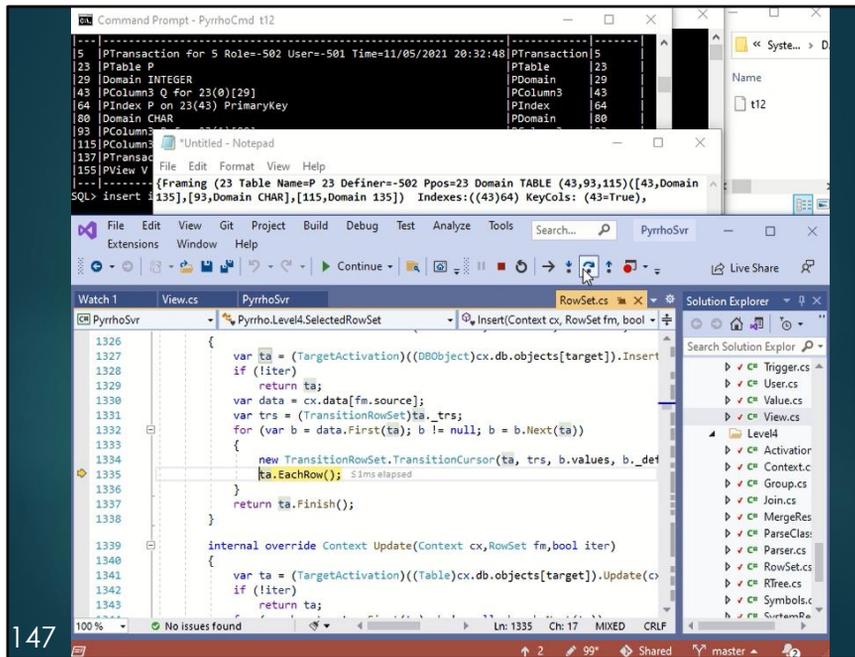
Step Over some more, to get back to EachRow.



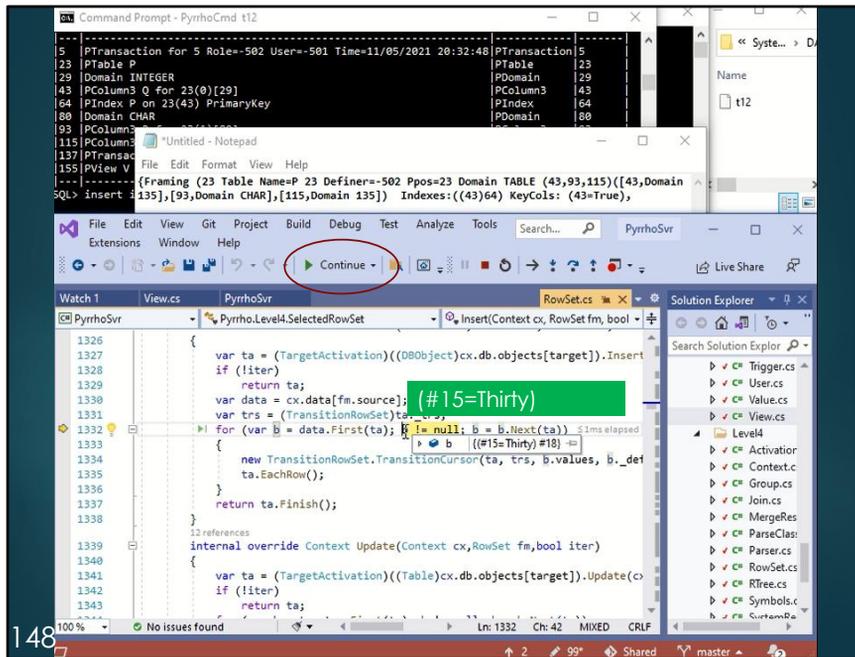
In `EachRow()` we have just finished `Add` for our Record. Step over to line 417.



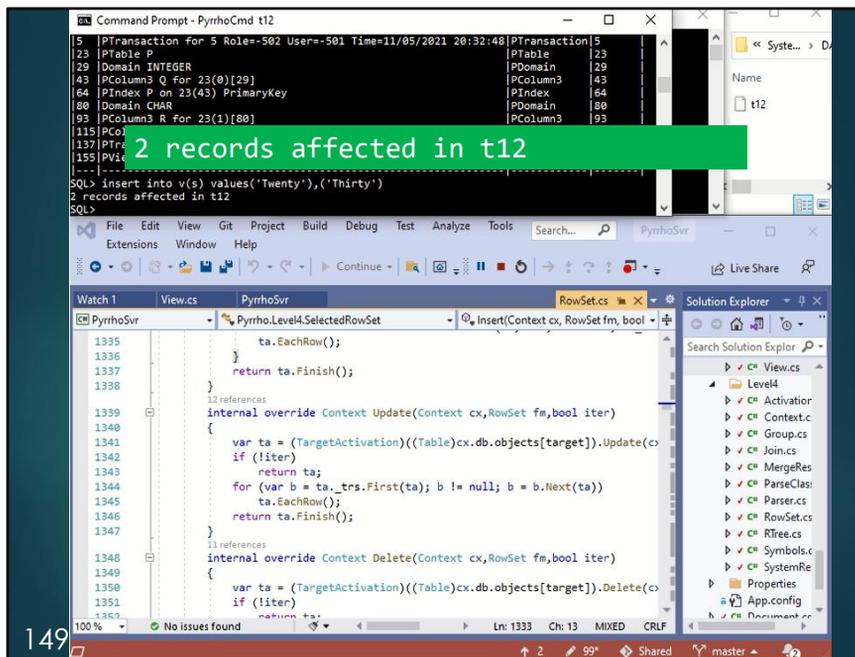
At line 417 we copy our transaction from the TargetActivation into the parent Context. (Just an assignment of a 64-bit pointer.)



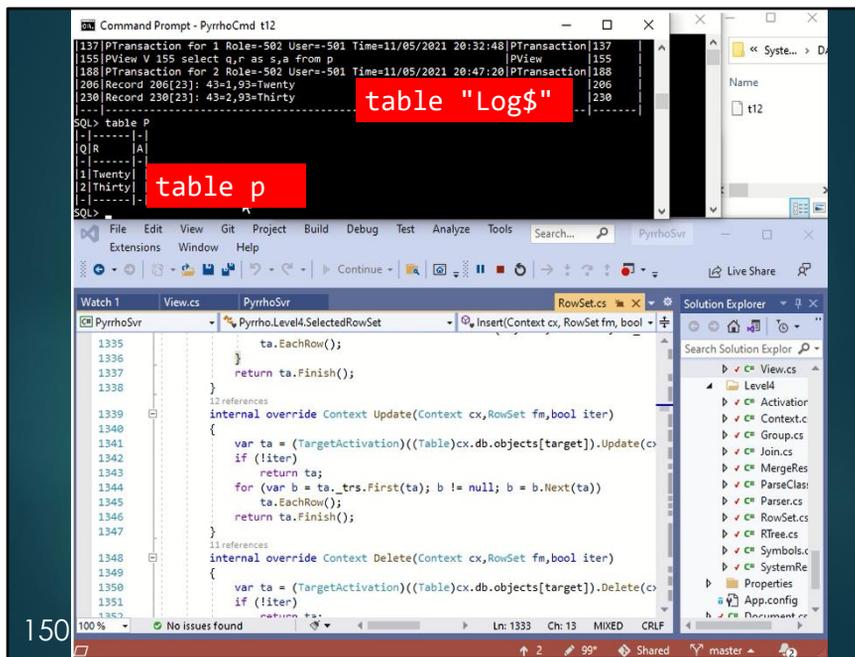
That finishes EachRow for this row of the data. Step Over to see the Cursor for the next row of the data



Hovering over `b`, we recognise the value `Thirty` here. The same thing will happen for this row, and then the transaction will `Commit()`, so just click `Continue` to finish.



The successful commit is reported to the client.



Confirm the new contents of the table P (this is the real target of the INSERT statement, not V!)

This completes the demonstration.

## Accessing Remote Data



- ▶ Using SQL to access a REST service
  - ▶ Requires a little extra syntax  
CREATE VIEW id OF rowtype AS GET url
  - ▶ The rowtype is like a table definition
    - ▶ Column names and domains
    - ▶ REST data received gets coerced to this type
  - ▶ The url is supplied as metadata
    - ▶ Maybe with other properties, e.g. MIME, AGENT
  - ▶ Could form simple joins with local data
- 15 ▶ We call this a RESTView

In the last part of this tutorial we want to demonstrate how these ideas can be extended to remote views (often called View-Mediated data Integration).

If we define a remote view, we need to give the domains of the columns (like in a table definition) that the remote data received from a REST service can be coerced to.

We also need to be told how to access it, e.g. a URL, MIME, locale. This can be done with metadata.

Some extra syntax needs to be added to SQL. We call this RESTView.

## Transactions and RESTview



- ▶ Transactions then need some thought
- ▶ RFC 7232 is helpful, as it explicitly covers the lost update problem
  - ▶ It proposes all HTTP1.1 services return entity-tags
  - ▶ These can be used for conditional requests
  - ▶ We can use these in the RESTView implementation
- ▶ But very few external REST services comply
  - ▶ Pyrrho has a suitable service that does
  - ▶ An interface called RESTIf tailored for other DBMS

152

Transactions then need some thought

RFC 7232 is helpful, as it explicitly covers the lost update problem

It proposes all HTTP1.1 services return entity-tags

These can be used for conditional requests

We can use these in the RESTView implementation

But very few real REST services comply

So Pyrrho has a suitable service that does

## REST in a branch transaction

- ▶ `http[s]://host[:port]/database/role[/table]`
- ▶ Implemented using HEAD and POST verbs
  - ▶ Sequence of ;-separated SQL statements
  - ▶ Changes POSTed at the end of the transaction
- ▶ Transaction control uses
  - ▶ If-Unmodified-Since for transaction start point
  - ▶ If-Match validates ETags, detects conflicts
- ▶ The REST service uses normal SQL statements
  - ▶ Pyrrho generates HTTP requests from these
  - ▶ Wherever remote views are referenced
  - ▶ Rewrites requests to reduce traffic (as in demo 3)
    - ▶ During the RowSet review stage of processing

153

Pyrrho offers two styles of interaction over HTTP, one using URL-based discrete operations, and another using POST to send a sequence of SQL statements to a remote database.

In both cases any changes should be done as a branch transaction at the commit point of the main transaction. But as we will see, we accumulate information about the steps as we go (like read constraints in demo 2), using the HEAD verb to anticipate the actual change. The information is exchanged using ETags, described next.

The REST service is more natural for an SQL-based client, and the rewriting process can take account of different SQL dialects, with metadata about the view identifying the SQL agent providing the service.

## Pyrrho's ETags



- ▶ ETags are defined in RFC7232
  - ▶ "\*" is an empty ETag
  - ▶ Anything else is opaque
  - ▶ Generated by the service provider
  - ▶ Not understood by anyone else
- ▶ Pyrrho's have form

*Table\_uid (' , RowDefPos\_uid ' , FilePos\_uid )+*

- ▶ Gives a sequence of specific rows
  - ▶ Defining position and latest position for each
- ▶ Or -1 for the whole table

154 ▶ And the highwater-mark for the table

ETags are defined in RFC7232. "\*" is an empty ETag, and anything else is opaque, that is, it is generated by the service provider, and not understood by anyone else.

These slides show some simple examples of Pyrrho's ETags. They have the form of a comma-separated list of integers:

It begins with `Table_uid` and this is followed by one or more pairs `RowDefPos_uid`, `FilePos_uid`.

Thus an ETag can specify (a) a sequence of specific row occurrences, with the defining position and latest position for each, or (b) -1 as a wildcard standing for all rows, and the highwater-mark for the table

## Demo 4: REST Views



- ▶ For example

```
[create view W of (e int,f char,g char)
as get etag http://localhost:8180/A/A/D]
```

- ▶ We will see this is an updatable view!
  - ▶ etag: use the RFC 7232 protocol
- ▶ We use Pyrrho's transactional REST service
- ▶ Start this up on the command line:

```
PyrrhoSvr -d:\DATA +s -H
```

- ▶ -H helps to monitor the HTTP traffic

155

In this demonstration we see how the framework developed in this tutorial applies when the data is remote.

In particular, we demonstrate an updatable RESTView where Pyrrho is used for both server and client (over HTTP, so could be remote).

```
[create view W of
(e int,f char, g
```

```
char) as get etag  
http://localhost:8  
180/A/A/D]
```

We will see that  
this is an  
updatable view.

The etag metadata  
flag tells Pyrrho  
to use RFC 7232.

To switch on Pyrrho's REST service, we add the `+s` flag. It starts an HTTP server (on port 8180 by default).

The `-H` flag gets diagnostic information in the server window showing HTTP requests and responses.

## Models Peer-to-Peer Banking

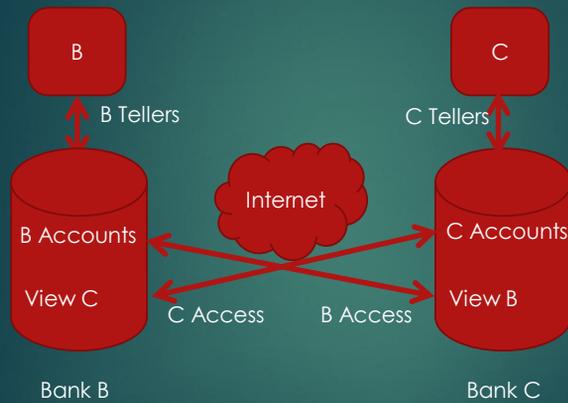


- ▶ Each bank has Roles for Tellers and Access
  - ▶ Access role can make transfers
  - ▶ Each bank defines its own accounts
    - ▶ Using the Teller Role
    - ▶ And enforces serializable transactions
  - ▶ Has updatable view for other banks
    - ▶ Using the Access role
    - ▶ Uses RFC 7232 to ensure ACID
  - ▶ For the demo it is kept simple: 2 banks, 1 Server, 1 user identity; but still use Internet
- 156

We can think of it as a model of peer-to-peer banking, with roles defined for local accounts and external transfers. Each bank is autonomous, and serializes transactions on their own accounts in the ordinary database way. For remote accounts they use ETags and RFC 7232 conditional requests. There could be dozens of banks, and many users granted these roles.

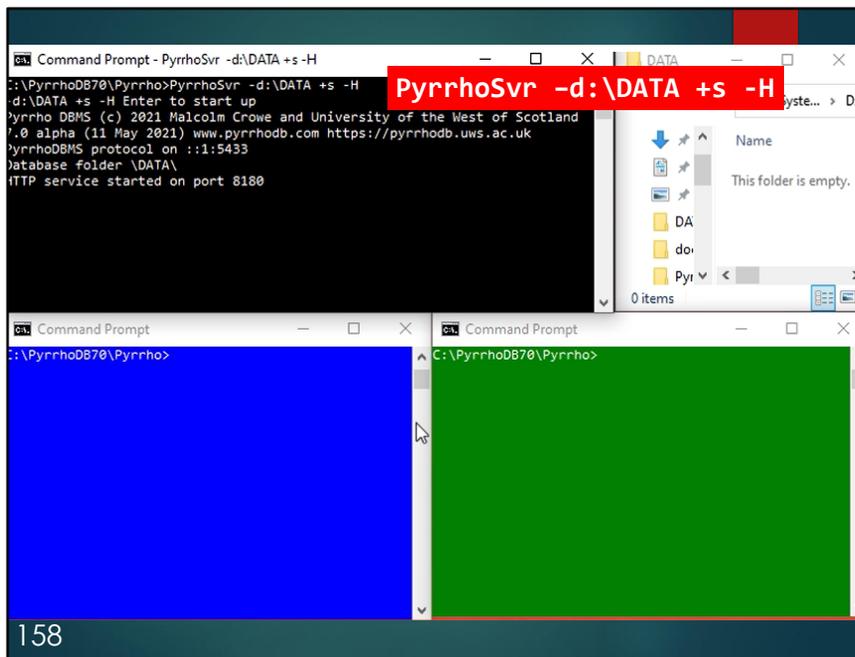
Users would not actually execute individual SQL statements but use banking applications that manage auditing, journaling, compliance etc.

## Uses Internet for Access



157

Each bank defines its own accounts, and creates a view of other banks. Views provide only a subset of information (e.g. can validate an account number) and can enable transfers of money in both directions. The diagram shows only a subset of the functionality required of the banking application: just enough to demonstrate the use of the ETag mechanism for transfers between local and remote accounts.



The scenario in this demo is a bank transfer between two banks.

This time the PyrrhoSvr needs to be started up with the command line shown:

**PyrrhoSvr -d:\DATA +s -H**

We check the database folder is empty and provide blue and green windows for the two bank clients.

```
PyrrhoCmd DB
C:\PyrrhoDB70\Pyrrho> create table AB(id int, bal int)
C:\PyrrhoDB70\Pyrrho> insert into AB values (100,1000),(110,1400)
C:\PyrrhoDB70\Pyrrho> create role DB
C:\PyrrhoDB70\Pyrrho> grant DB to "current username"
C:\PyrrhoDB70\Pyrrho> create view RC of (id int, bal int) as get etag milli 'http://../AC'
```

159

To fit things on the slide we use short identifiers: AB for accounts in bank B, etc. We create an accounts table with two accounts, id 100 with balance 1000, and id 110 with balance 1400.

To make this table accessible over the Internet at least one role and user needs to be defined, so here we use the current user on the device.

And we prepare a remote view for us to access the remote accounts AC in bank C.

The metadata flags etag and milli tell Pyrrho to enforce RFC 7232 conditional request protocol, but using millisecond accuracy for the If-Unmodified-Since header.

```
Command Prompt - PyrrhoCmd DC
C:\PyrrhoDB70\Pyrrho>PyrrhoSv
-d:\DATA +s -H Enter to start
Pyrrho DBMS (c) 2021 Malcolm
7.0 alpha (11 May 2021) www.py
PyrrhoDBMS protocol on ::1:54
Database folder \DATA\
HTTP service started on port 8

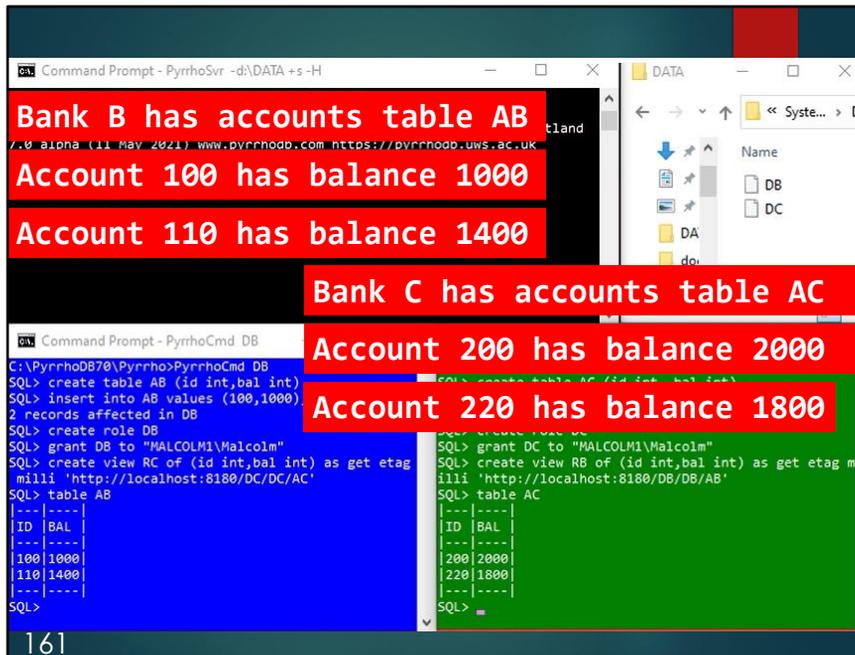
C:\PyrrhoDB70\Pyrrho>PyrrhoCmd DB
SQL> create table AB (id int,bal int)
SQL> insert into AB values (100,1000),(110,1400)
2 records affected in DB
SQL> create role DB
SQL> grant DB to "MALCOLM1\Malcolm"
SQL> create view RC of (id int,bal int) as get etag
milli 'http://localhost:8180/DC/DC/AC'
SQL>

C:\PyrrhoDB70\Pyrrho>PyrrhoCmd DC
SQL> create table AC (id int, bal int)
SQL> insert into AC values (200,2000),(220,1800)
2 records affected in DC
SQL> create role DC
SQL> grant DC to "MALCOLM1\Malcolm"
SQL> create view RB of (id int,bal int) as get etag m
illi 'http://localhost:8180/DB/DB/AB'
SQL>
```

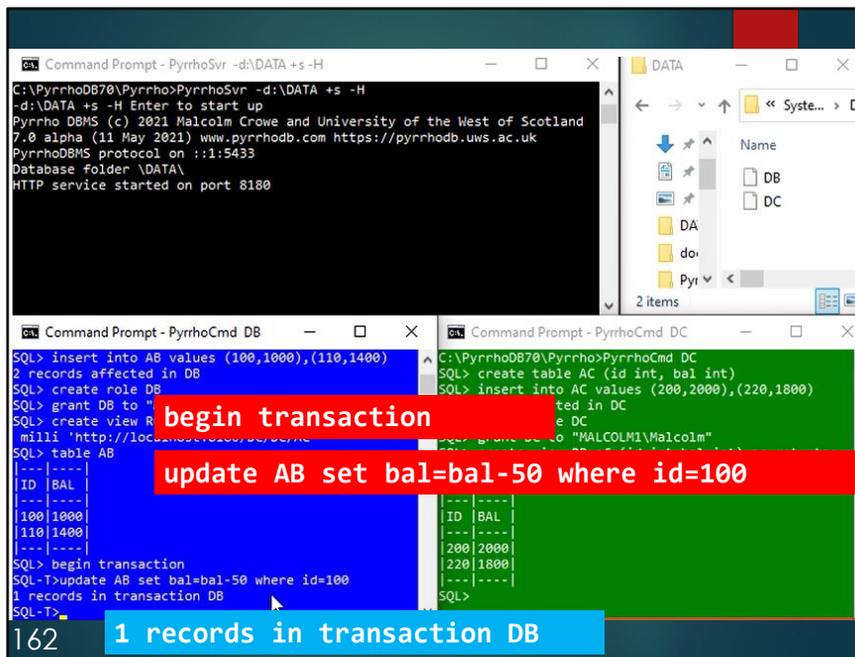
Bank C has a similar set-up.

Database DC is the database for bank C. A user is defined who is authorised to connect to role DC.

AC is the table of accounts in bank C, and the RESTView RB gives remote access to accounts in bank B.



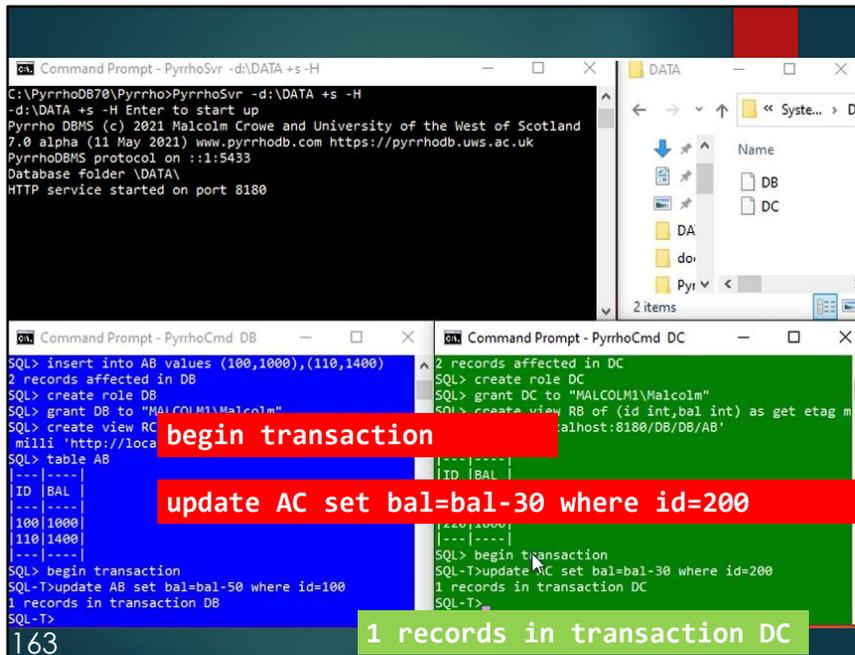
Let's show the current state of the accounts in the two banks.  
In the next steps, bank B will try to transfer £50 to account 200 in bank C, while bank C tries to transfer £30 from account 200 to an account in bank B.



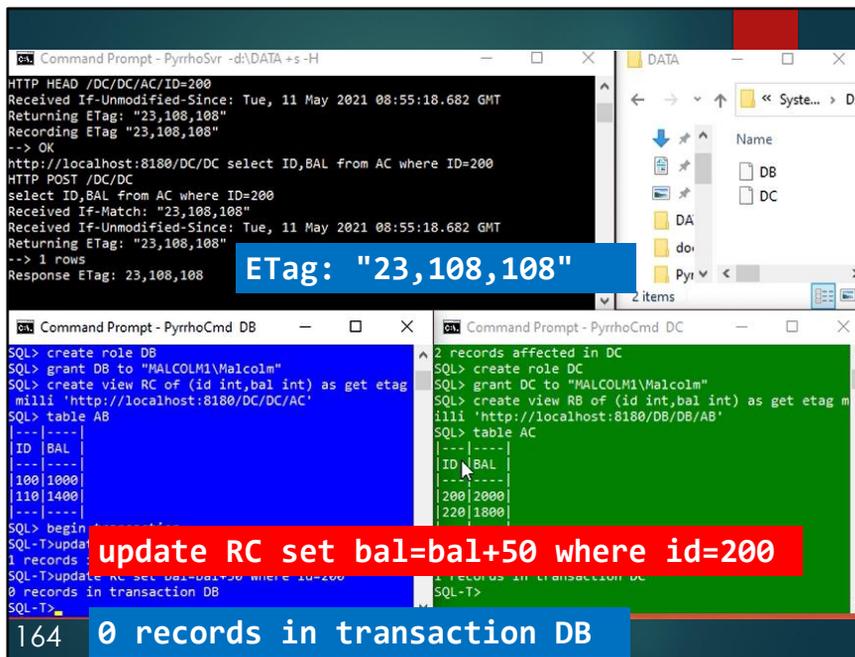
Two overlapping transactions will conflict on account 200.

In this slide, bank B starts a transaction and takes £50 from account 100 to be transferred. Nothing is written to disk yet.

The report on the update makes the point that the new record is in the transaction, rather than the database.



Someone in bank C starts a transaction around the same time and takes £30 from account 200 for another transfer.



Now the blue teller transfers the £50 to account 200 in bank C using the remote view RC of their accounts. This is still in the transaction, and nothing has been committed. However we see Internet activity as bank B checks that 200 is an account in bank C. we suppose that the balance data will be withheld, because of the permissions, but bank B does receive an ETag. The response on bank B indicates that the changes are not for the local bank.

It is hoped that COUNT(\*) can (always) be used instead of selecting columns to create ETags: will become clearer during the next phase of implementation.

```

Command Prompt - PyrhoSvr - d:\DATA +s -H
HTTP HEAD /DB/DB/AB/ID=110
Received If-Unmodified-Since: Tue, 11 May 2021 08:55:30.345 GMT
Returning ETag: "23,126,126"
Recording ETag "23,126,126"
--> OK
http://localhost:8180/DB/DB select ID,BAL from AB where ID=110
HTTP POST /DB/DB
select ID,BAL from AB where ID=110
Received If-Match: "23,126,126"
Received If-Unmodified-Since: Tue, 11 May 2021 08:55:30.345 GMT
Returning ETag: "23,126,126"
--> 1 rows
Response ETag: 23,126,126

DATA
Name
DB
DC
DA
do
Pyri

Command Prompt - PyrhoCmd DB
SQL> create role DB
SQL> grant DB to "MALCOLM\Malcolm"
SQL> create view RC of (id int,bal int) as get etag milli 'http://localhost:8180/DC/DC/AC'
SQL> table AB
ID |BAL|
---|---|
100|1000|
110|1400|
---|---|
SQL> begin transaction
SQL-T>update AB set bal=bal-50 where id=100
1 records in transaction DB
SQL-T>update RC set bal=bal+50 where id=200
0 records in transaction DB
SQL-T>

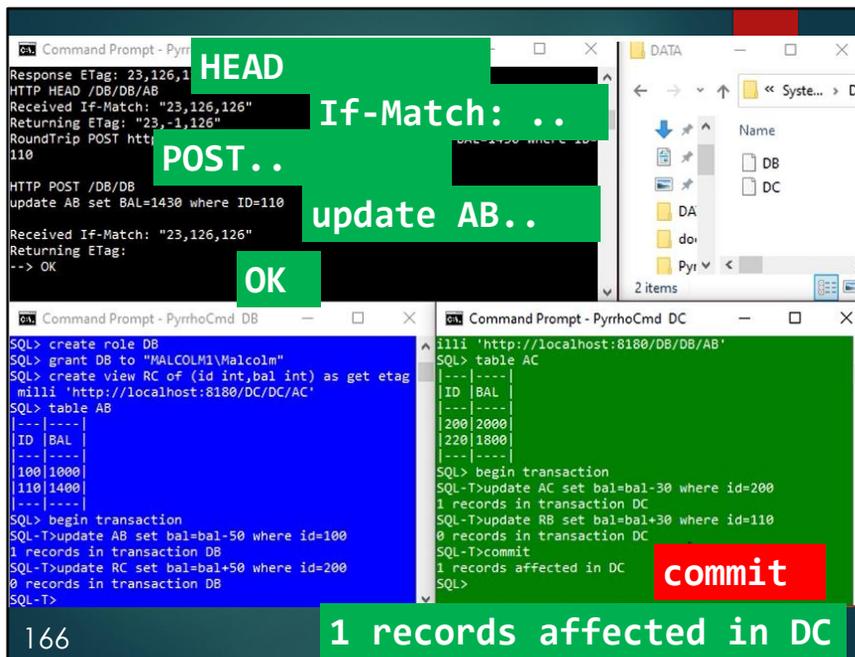
165

Command Prompt - PyrhoCmd DC
SQL> grant DC to "MALCOLM\Malcolm"
SQL> create view RB of (id int,bal int) as get etag milli 'http://localhost:8180/DB/DB/AB'
SQL> table AC
ID |BAL|
---|---|
SQL> begin transaction
SQL-T>update AC set bal=bal-30 where id=200
1 records in transaction DC
SQL-T>update RB set bal=bal+30 where id=110
0 records in transaction DC
SQL-T>

0 records in transaction DC

```

The green teller also starts a transfer, of his £30 from account 200 to account 110, and receives an ETag for remote account 110. At this point both tellers are about to commit their transaction. The first to do so will succeed, but no changes have been made yet.



The green teller gets there first, and commits their transaction.

The first step is a HEAD request to verify the ETag sent to the green database is still valid. (It is.)

Then the update is POSTed to bank B, and on receiving OK from DB, the change is committed to local account 200 in bank C.

This will invalidate the ETag previously sent to the blue client.

The response to the commit indicates that the change to local account 200 is now recorded durably in the database.

```

Command Prompt - PyrrhoSvr - d:\DATA +s -H
Received If-Match: "23,126,126"
Returning ETag: "23,-1,126"
RoundTrip POST http://localhost:8180/DB/DB update AB set BAL=1430 where ID=110

HTTP POST /DB/DB
update AB set BAL=1430 where ID=110

Received If-Match: "23,126,126"
Returning ETag:
--> OK
HTTP HEAD /DC/DC/AC
Received If-Match: "23,108,108"

Command Prompt - PyrrhoCmd DC
SQL> create view RC of (id int,bal int) as get etag milli 'http://localhost:8180/DC/DC/AC'
SQL> table AB
----|----|
ID |BAL |
----|----|
100|1000|
110|1400|
----|----|
SQL> begin transaction
SQL-T>update AB set bal=bal-50 where id=100
1 records in transaction DB
SQL-T>update RC set bal=bal+50 where id=200
0 records in transaction DB
SQL-T>commit
ETag validation failure
SQL>

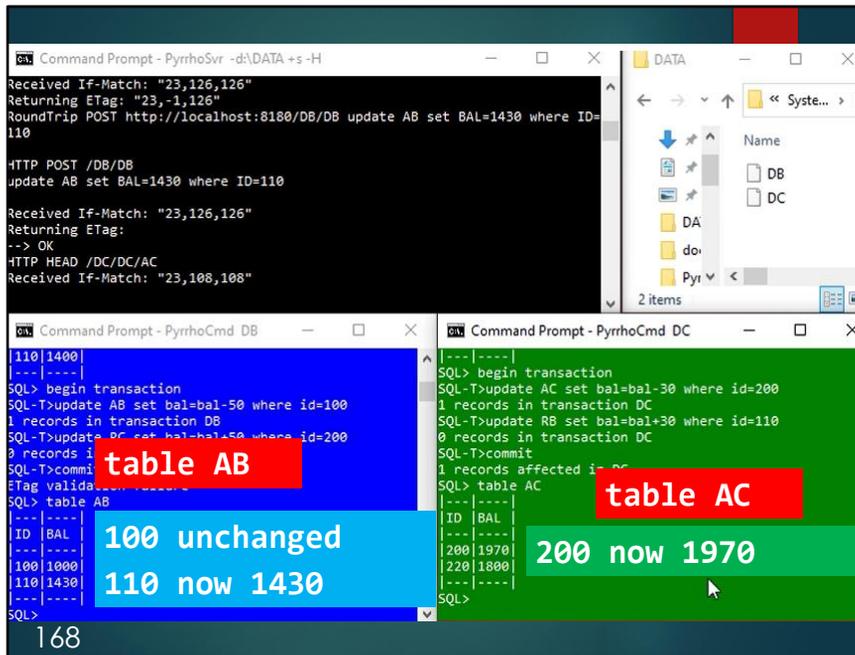
167

Command Prompt - PyrrhoCmd DC
llli 'http://localhost:8180/DB/DB/AB'
SQL> table AC
----|----|
ID |BAL |
----|----|
200|2000|
220|1800|
----|----|
SQL> begin transaction
SQL-T>update AC set bal=bal-30 where id=200
1 records in transaction DC
SQL-T>update RB set bal=bal+30 where id=110
0 records in transaction DC
SQL-T>commit
1 records affected in DC
SQL>

```

Sure enough, when the blue client attempts to commit, and sends its change to bank C, bank C reports that the ETag is invalid, and the transaction is aborted. The failure of this HEAD roundtrip rolls back the blue transaction entirely.

In this case, if the green client committed first, the local update would fail and a normal (non-ETag based) transaction conflict condition would be raised.



The teller in bank B can verify that the transfer from account 100 has not occurred, while bank C's transaction transferring £30 has been committed in both databases.

## Demo 4 Extra: RowSet Review



▶ A view of a local join with a remote view (it is updatable!)

▶ Database A:

```
create table D(e int primary key,f char,g char)
```

```
insert into D values (1,'Joe','Soap'), (2,'Betty','Boop')
```

```
create role A; grant A to "MALCOLM1\Malcolm"
```

▶ Database B:

```
create view W of (e int, f char, g char) as get  
etag milli 'http://localhost:8180/A/A/D'
```

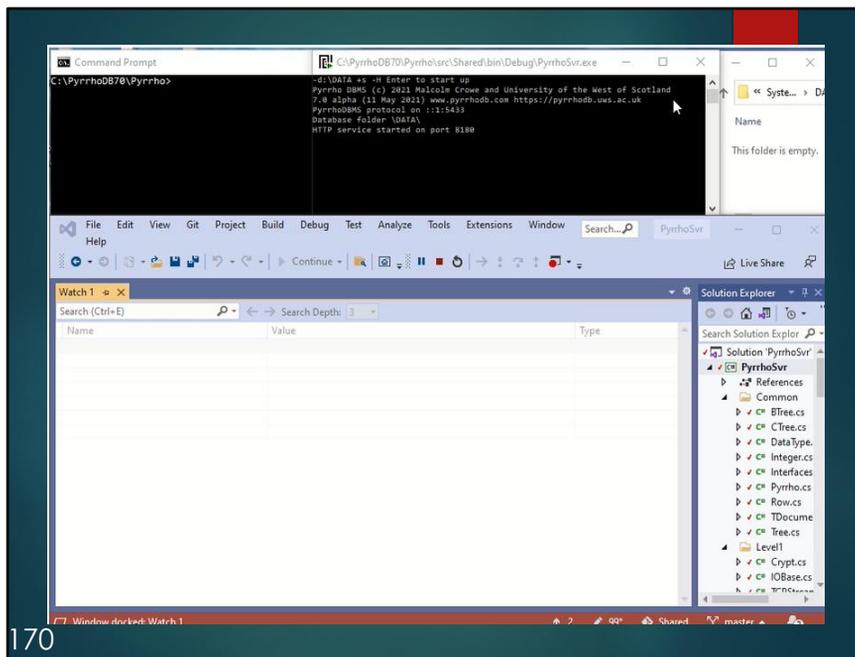
```
create table H (e int primary key, k char, m int)
```

```
insert into H values (1,'Cleaner',12500), (2,'Manager',31400)
```

```
create view V as select * from W natural join H
```

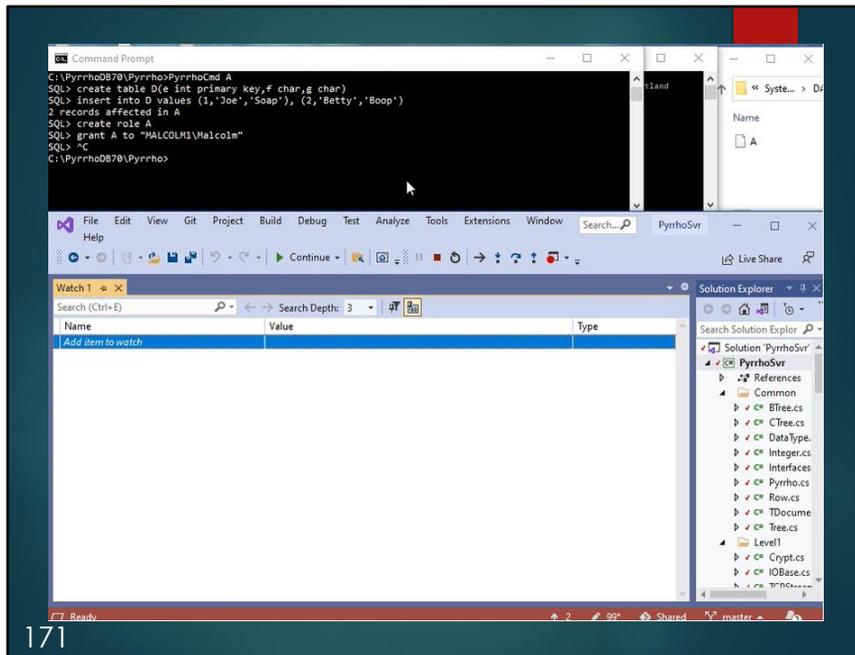
169

As an extra demo, we will use Visual Studio to explore RowSet Review, using as an example a viewed join of a local table with a remote view. This is part of the test23 suite in the Pyrrho V7 alpha distribution, where it is shown to be updatable. The example is a simplified system of a company employee database that records job titles and salaries, but where the employees' names are stored in another database.

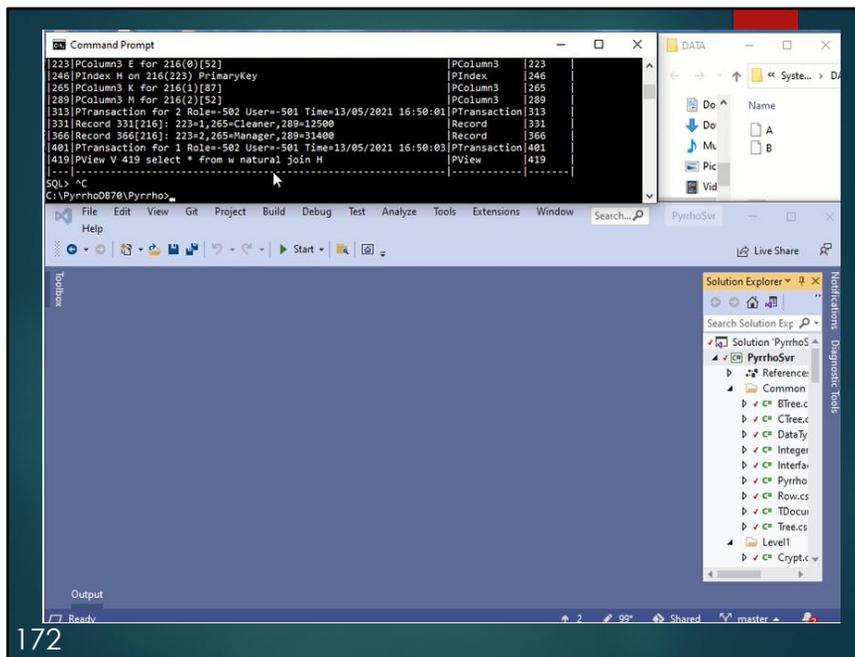


170

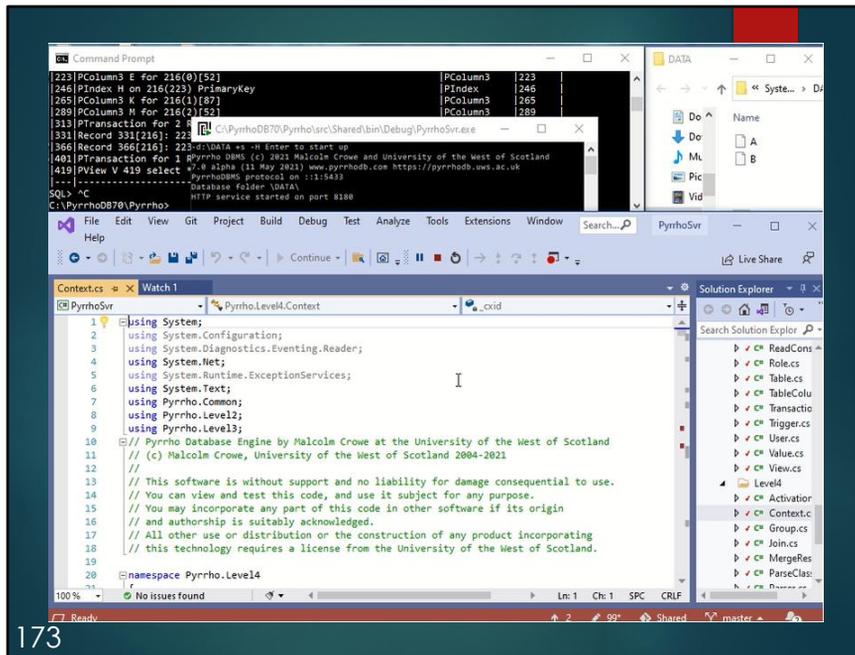
We start as before with a debug session on Visual Studio where the command line arguments are again `-d:\DATA +s -H`. The top row shows a command window for the clients, the debugger's server window, a glimpse of a file manager showing the empty database folder.



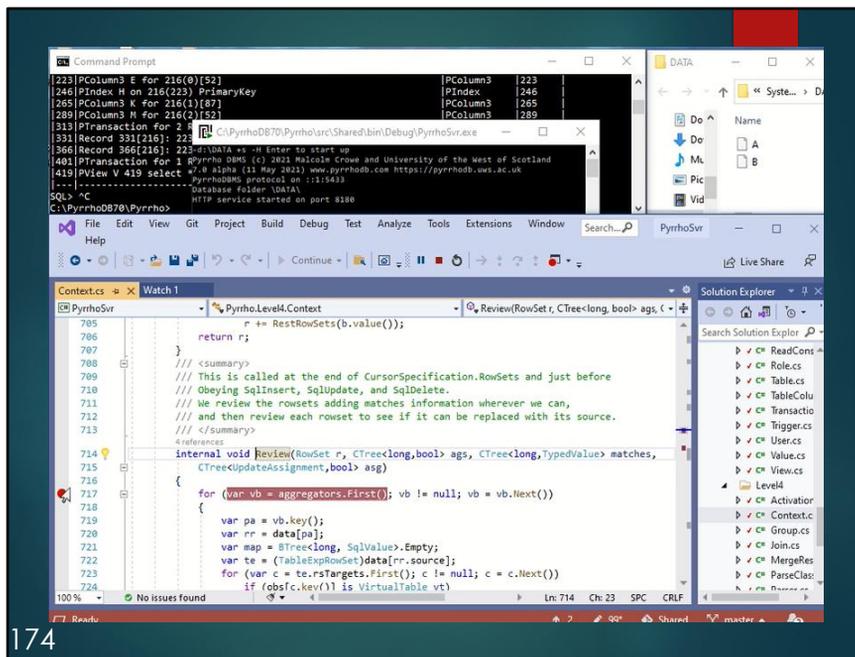
We first create database A with the SQL statements on slide 169.



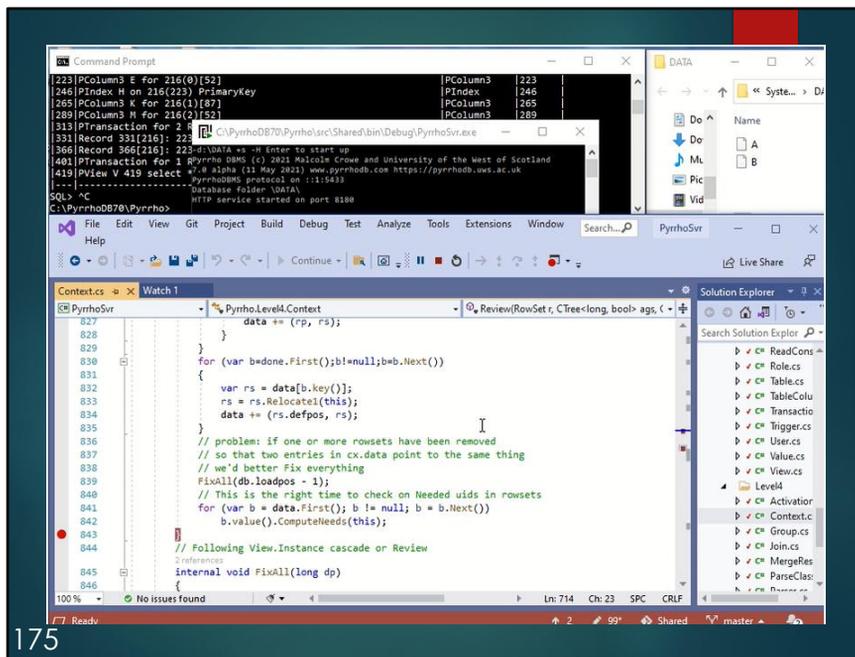
Now create the database B using the statements on slide 169. We have also had table "Log\$" from which we see that view V is at position 419.  
 Stop the server: we do this for reproducibility, as compiled object uids are different during the session in which objects are created. Restart the server.



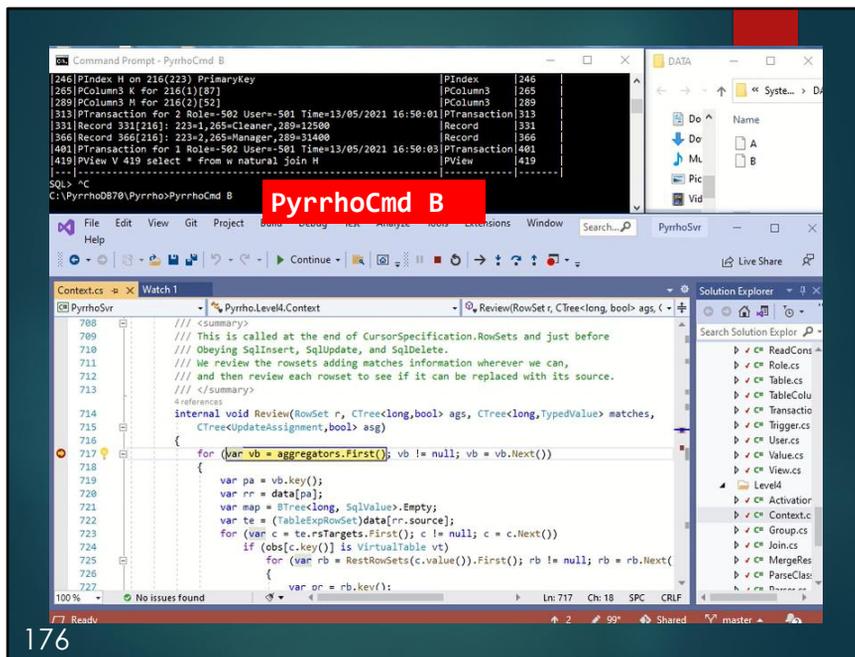
Find Context.cx in Level4 and double-click it. We are going to place breakpoints in the Context.Review() method.



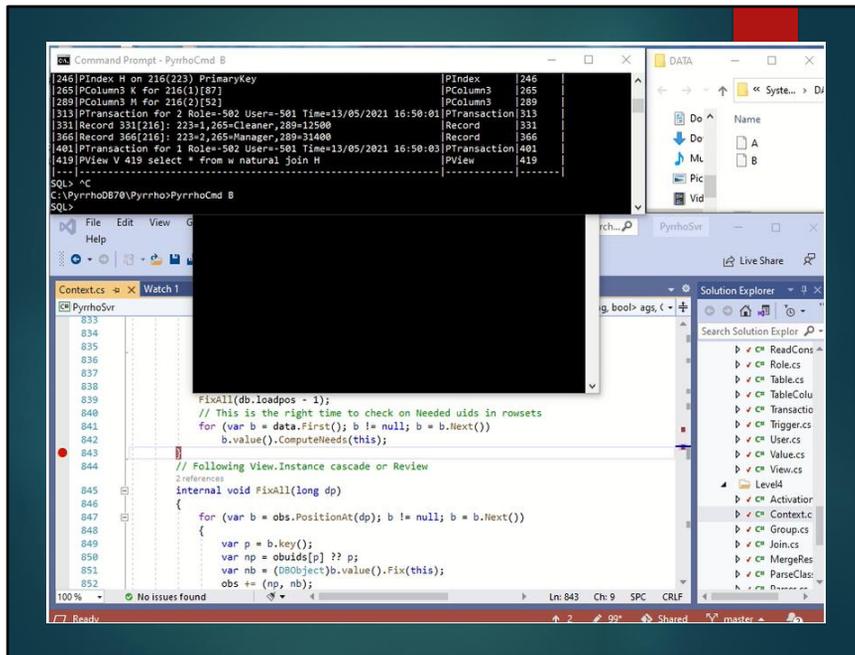
The breakpoint for the start of Context.Review() is at line 717 in Context.cs.



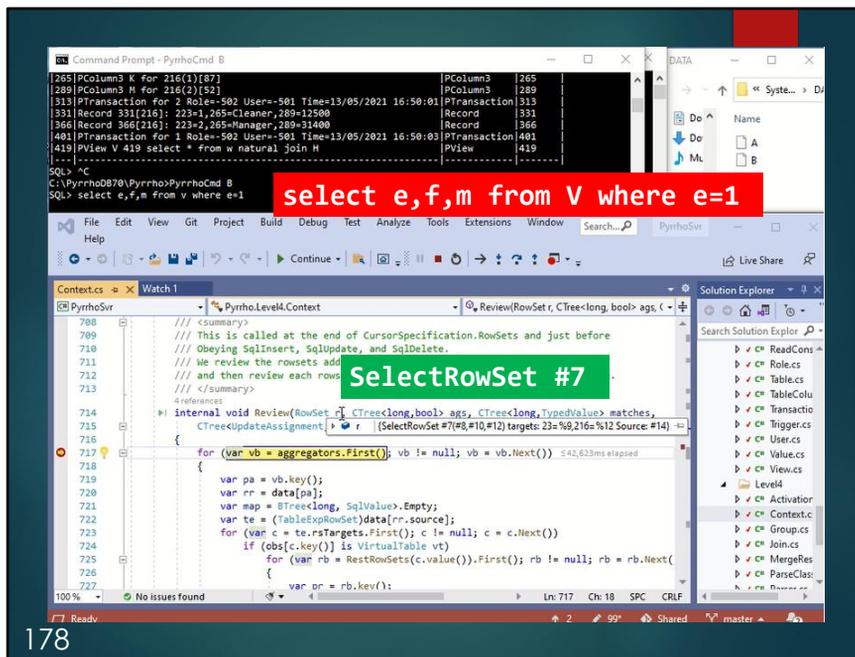
The breakpoint at the end of the `Context.Review()` method is at line 843 of `Context.cs`.



When we reopen database B, it is loaded into the server, and the compilation process for the view calls `Context.Review()`  
 Click Continue twice to get the normal `SQL>` prompt.



We are going to select some items from V.

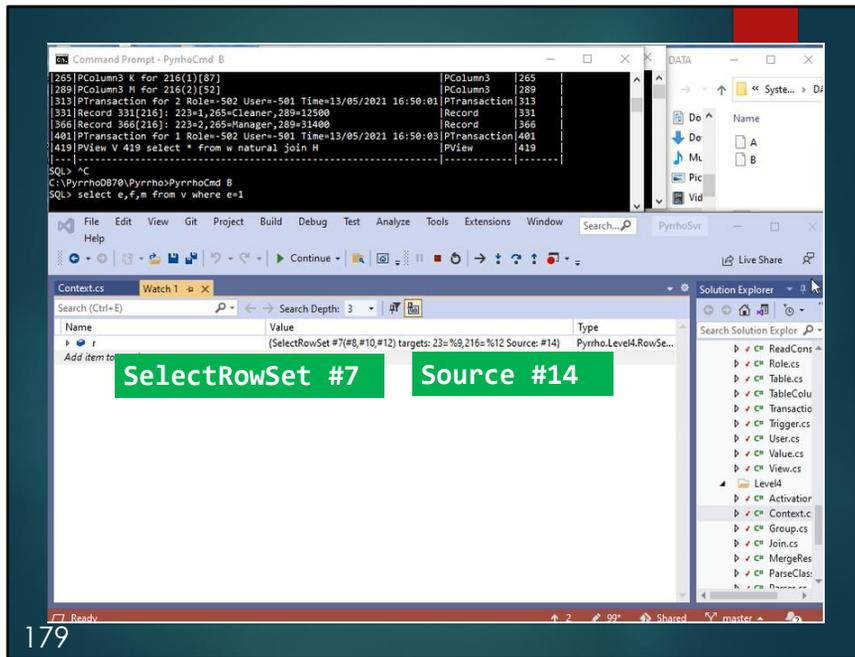


The given select statement is **select e,f,m from V where e=1**.

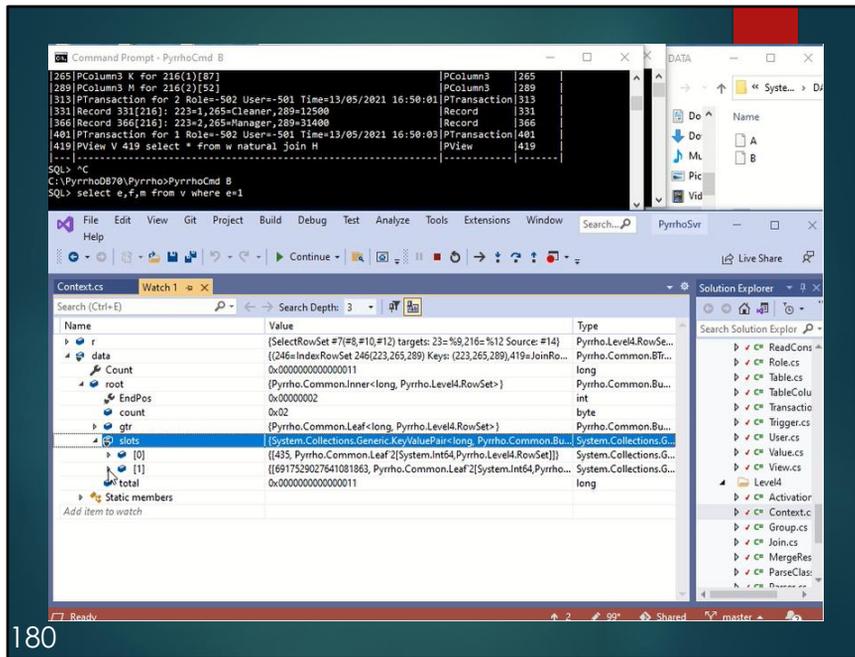
(Note the position of e in this statement: it occurs at position 8, so and the equals sign of the where condition is at position 28. We will see these numbers later.)

RowSet Review for the select statement stops at the breakpoint we set.

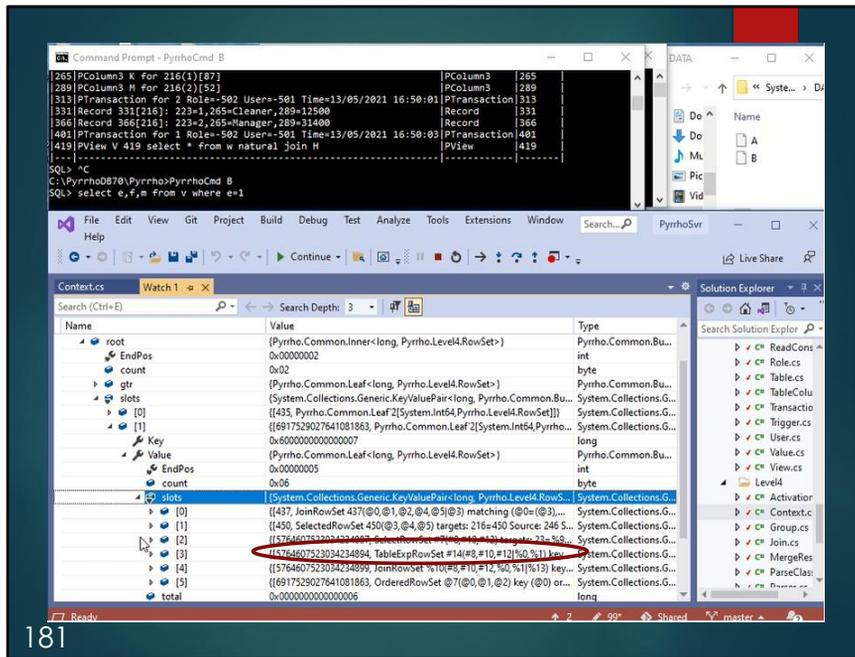
We are going to review the given RowSet r and its sources using the Watch window. It is a SelectRowSet.



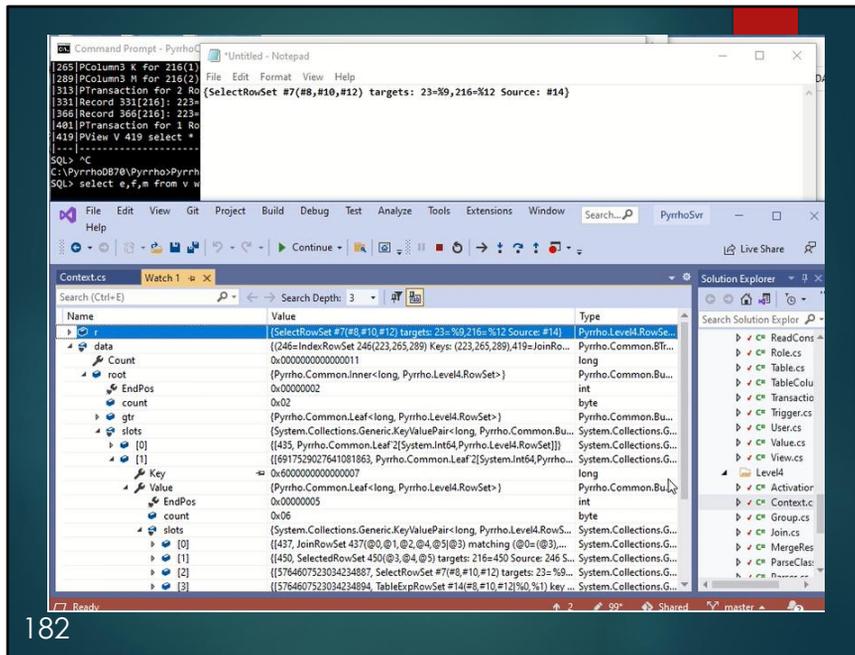
We see the source is a rowSet #14 . To examine other rowSets we need to look at the **data** BTree in Context.



Begin to expand **data**. RowSet #14 will be near the middle. Expand node root.slots[1].

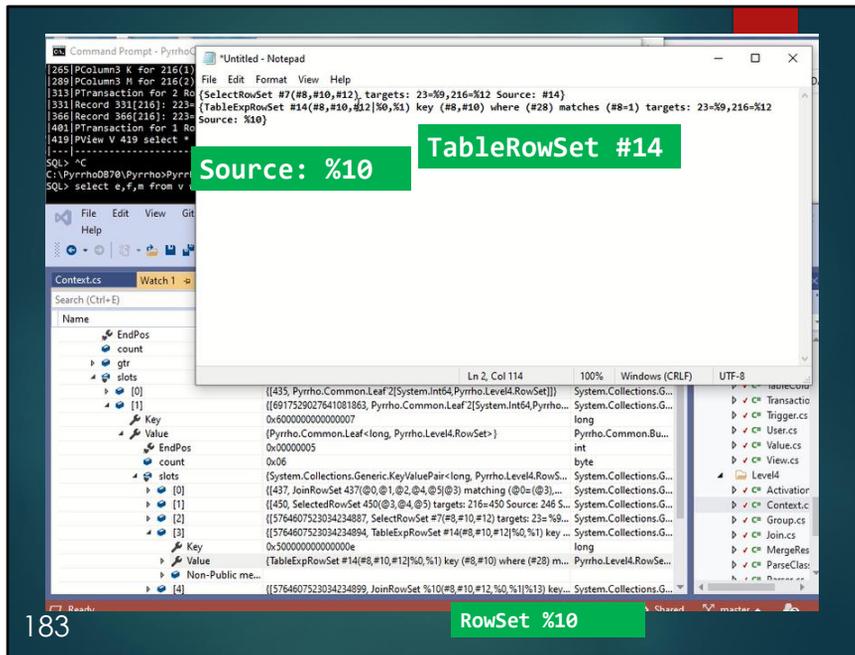


Expand the node containing TableRowSet #14

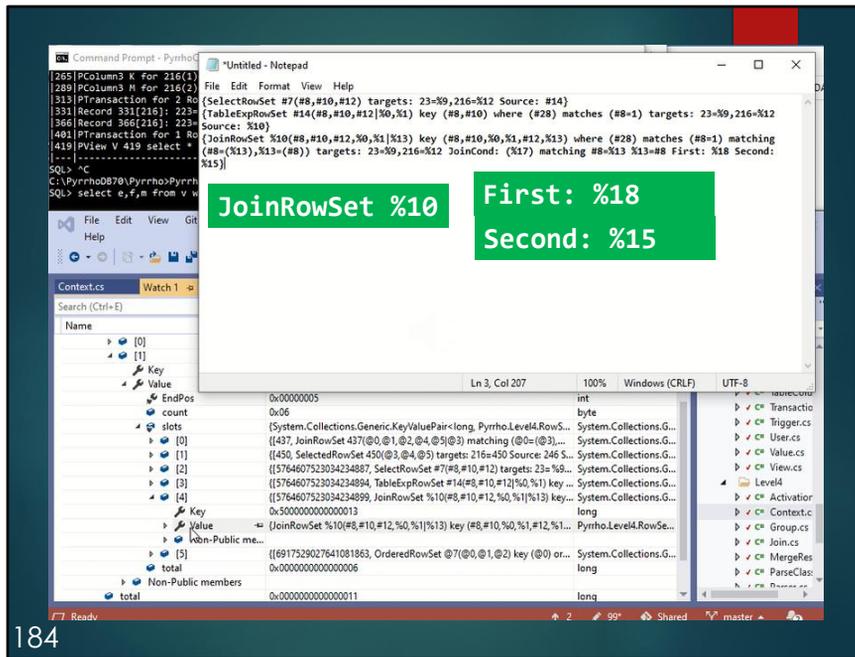


182

Open a Notepad window. Scroll to the top and copy the value of r into the Notepad.

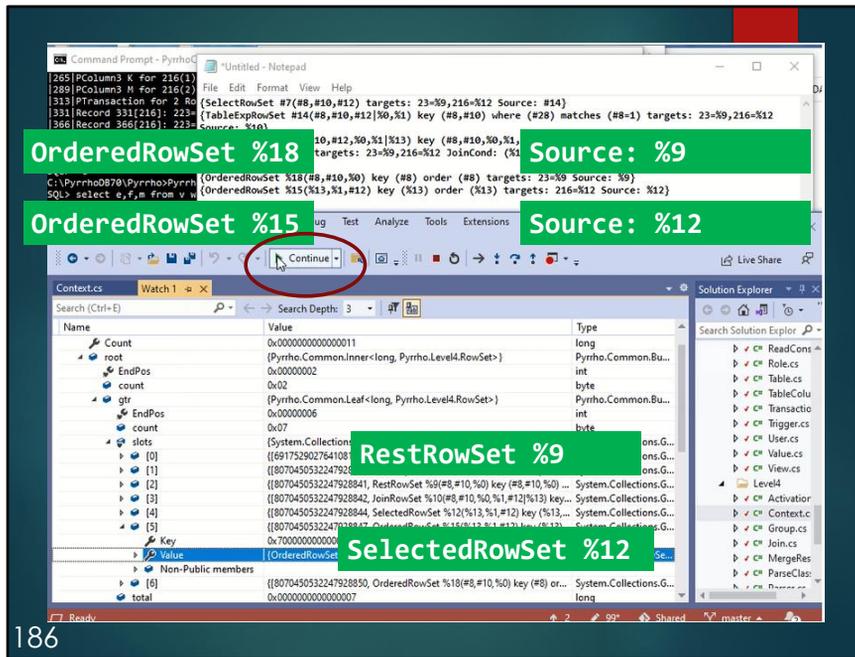


Expand the node containing the TableResultSet #14 and copy its value into the Notepad. We see its source is RowSet %10, which we can see in the next node below.

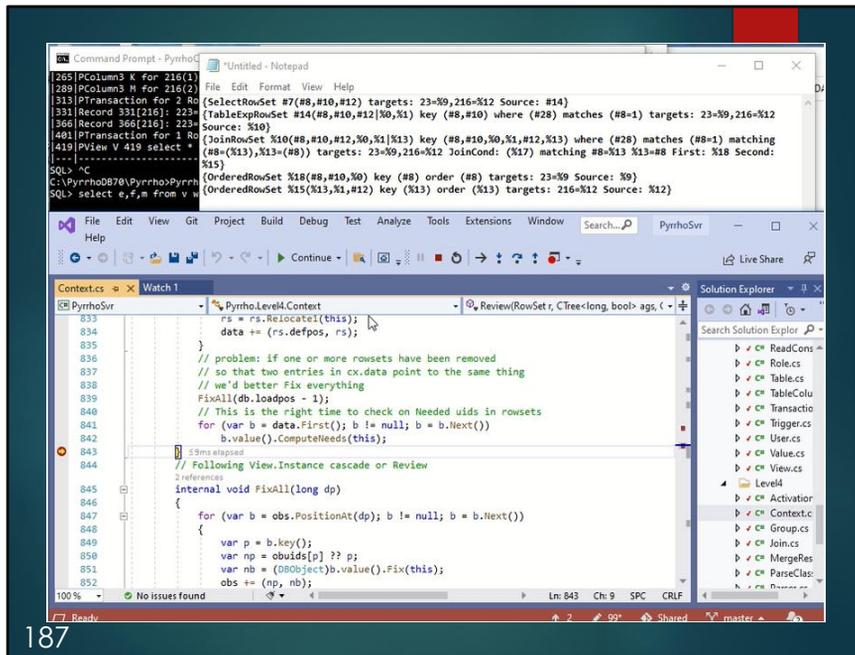


It is a JoinRowSet, with sources %18 and %15. These will both be in the **gtr** node of the **data** tree.

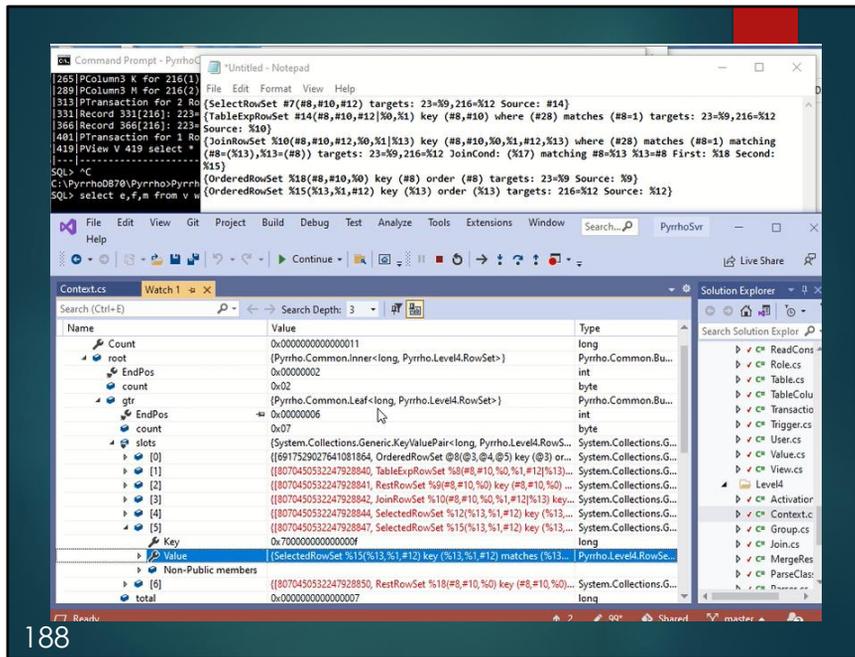




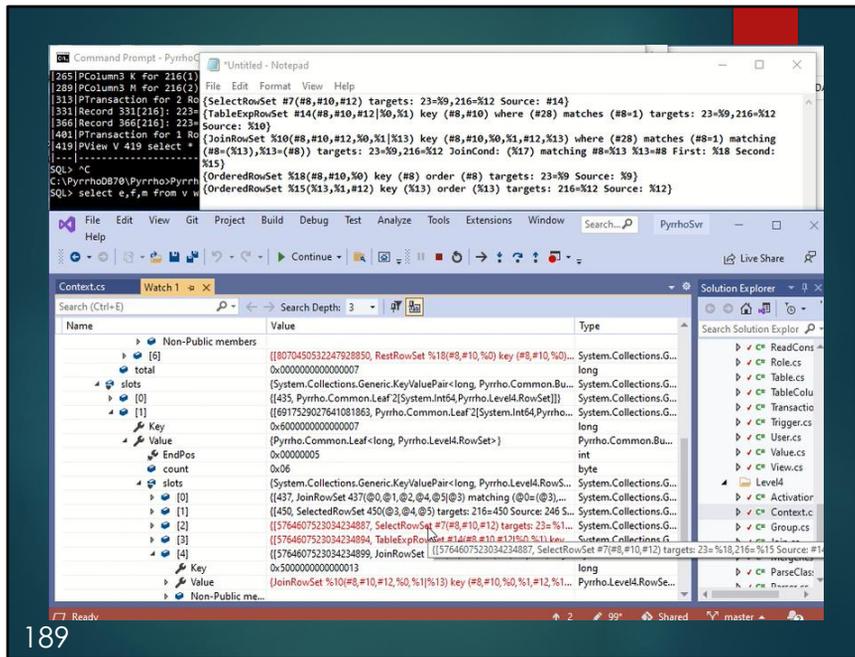
It is time for a short cut here: %18 has source %9 which we can see is a RestRowSet, and %15 has source %12, which is a SelectedRowSet. So let's pick them up later, at the end of the Review. Click Continue.



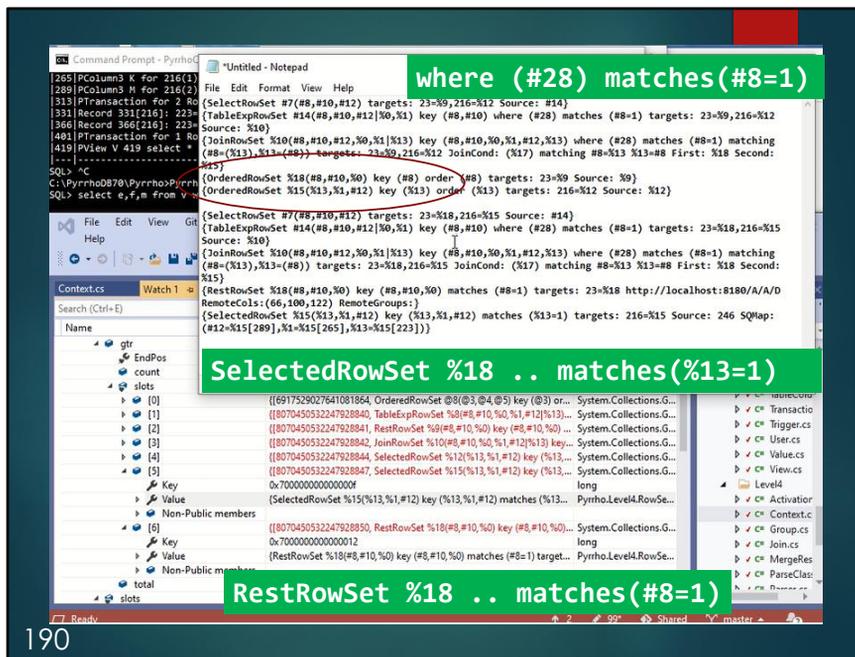
This is the breakpoint at the end of Review. Return to the Watch window.



Visual Studio reports changes to all of these RowSets. r will also be out of date, so repeat the above extraction from the data tree, starting with the modified SelectRowSet #7 (scroll down for it).



We see the changed version: copy its value to the Notepad and continue with its sources etc as we did before.



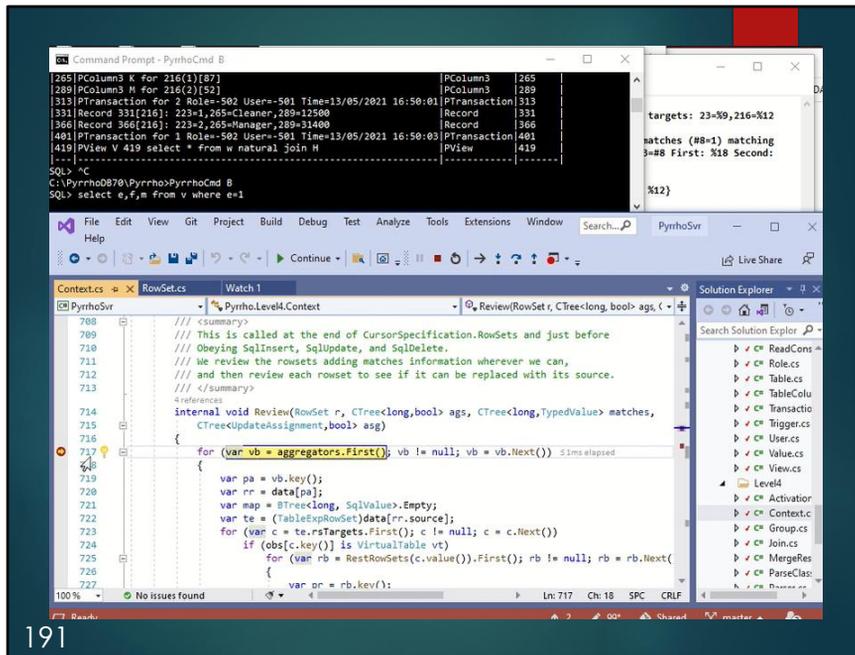
We see that a number of rowsets that were there before have simply been removed: there is no need for %9 or %12 and the OrderedRowSets %18 and %15 have been replaced by RestRowSet %18 and SelectedRowSet %15.

Importantly, the where condition `e=1` (where (#28) matches (#8=1)) has been analysed.

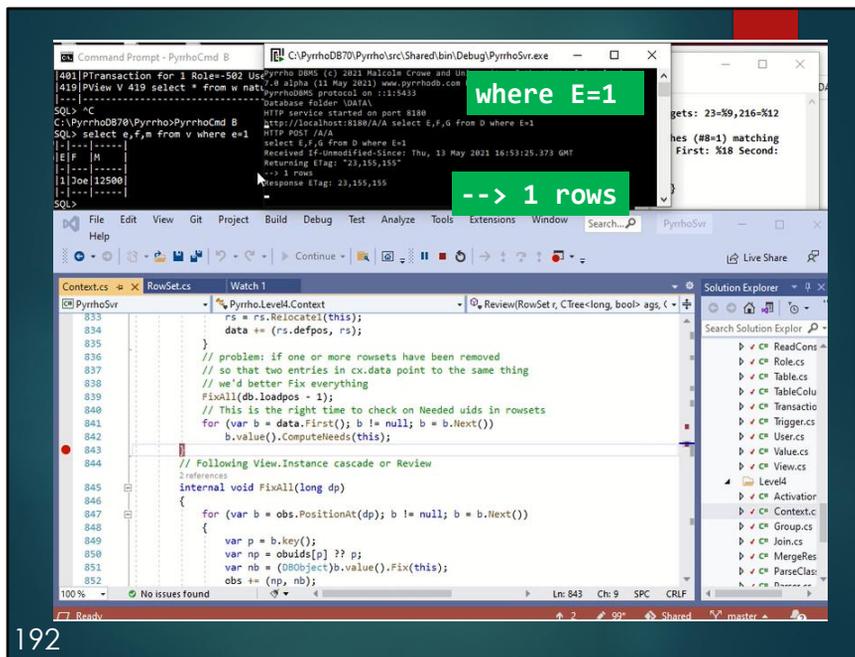
The Review process has noticed that the where-condition collapses both sides of the join to a single row, so that the orderedrowsets are not required.

And the RestRowSet now has the matches condition `#8=1` which will be included in the HttpRequest to the remote database.

Click Continue.



During the HttpService, our breakpoint gets hit again. Click Continue twice (before the timeout!),



We get the results of course, but importantly we see that only one row was requested from the remote database.  
 (More generally, aggregations can also result in only one row being returned.)  
 This concludes the demonstration of RowSet Review.

## Current Status



- ▶ Updatable simple RESTViews are completed
  - ▶ GET USING to be verified: uses JoinRowSets
  - ▶ Transaction implementation using RFC 7232
  - ▶ Combines SQL and HTTP, Data and Web services
- ▶ Next Steps in Pyrrho V7 experiment:
  - ▶ Extend RowSet Review to aggregations, groups
  - ▶ Verify Role-based security implementation
  - ▶ Extend to Versioned Object API for Web apps
  - ▶ Run the TPCC benchmark again

193

Pyrrho V7 is still at the alpha stage of development, and as this tutorial demonstrates, has a working implementation of simple updatable RESTViews. Inserts and Deletes have been implemented and tested. There is a more complex RESTView mechanism that managed a set of views with similar properties, implemented as a join. This implementation is still to be verified at the time of preparation of this tutorial, but it should follow from the implementation of updatable joins.

Once this is done, the plan is to extend RowSet Review to aggregations and groups with a more comprehensive set of tests. Next will come verification of the role-based security implementation, and the versioned Web API.

## Conclusions



- ▶ The approach is practical and safe
  - ▶ But the ideas are not really new (1982, 2014,..)
- ▶ Recommendations for DBMS implementers:
  - ▶ Use true serializable transactions
  - ▶ Use shareable data structures
  - ▶ Implement RowSets not Queries
  - ▶ Never compromise consistency or isolation

194

Current conclusions from the work are that the approach is practical and safe, and builds on ideas that have been around for a long time, though not in mainstream DBMS implementation.

DBMS implementers should adopt some or all of the following recommendations: serializable transactions, sharable data structures, rowSets not queries; but never compromise consistency or isolation.

## Questions?



- ▶ Please send any questions or comments not answered today
- ▶ [malcolm.crowe@uws.ac.uk](mailto:malcolm.crowe@uws.ac.uk)
- ▶ I will be running some live Zoom sessions after the conference
- ▶ (But ensure the Subject line of the email mentions Pyrrho)

195

I welcome discussion, and will be happy to provide interactive sessions on request.

## Further reading



- ▶ Crowe, M.K., Laux, F.: Reconsidering Optimistic Algorithms for Relational DBMS, DBKDA 2020
- ▶ Crowe, M.K., Matalonga, S., Laiho, M.: StrongDBMS: built from immutable components, DBKDA 2019
- ▶ Crowe, M.K., Fyffe, C: Benchmarking StrongDBMS, Keynote speech, DBKDA 2019
- ▶ The Pyrrho Manual, SourceIntro, source code on [Github.com/MalcolmCrowe/ShareableDataStructures/PyrrhoV7alpha](https://github.com/MalcolmCrowe/ShareableDataStructures/PyrrhoV7alpha)
- ▶ T. Krijnen, and G. L. T. Meertens, “Making B-Trees work for B”. Amsterdam: Stichting Mathematisch Centrum, 1982, Technical Report IW 219/
- ▶ <https://pyrrhodb/blogspot.com>
  - ▶ Esp: 2017 posts on adapter functions, RESTView